

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

驾驭文本

文本的发现、组织
和处理

Taming Text

How to Find, Organize,
and Manipulate It

Grant S. Ingersoll
[美] Thomas S. Morton 著
Andrew L. Farris

王斌 译



作者简介

Grant S. Ingersoll是一位工程师、讲师和培训师，也是Lucene代码的提交者以及机器学习项目Mahout的联合创始人。

Thomas S. Morton是OpenNLP和Maximum Entropy（最大熵）的主要开发者。

Andrew L. Farris是一位技术顾问、软件开发人员及Mahout、Lucene和Solr的贡献者。

驾驭文本

文本的发现、组织和处理

Taming Text

How to Find, Organize, and Manipulate It

Grant S. Ingersoll

[美] Thomas S. Morton 著

Andrew L. Farris

王斌 译

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

文本处理是目前互联网内容应用（如搜索引擎、推荐引擎）的关键技术。本书涵盖了文本处理概念和技术的多个方面，包括文本预处理、搜索、字符串匹配、信息抽取、命名实体识别、分类、聚类、标签生成、摘要、问答等。本书的特点在于通过实例来理解文本处理的这些概念和技术，读者利用现有的开源工具就可以自己实现这些实例。本书适合互联网文本内容处理领域的开发人员阅读，也适合有志于加入这一领域的学生、从业人员阅读。即使对于已经从事多年文本处理研究和开发工作的人员来说，本书也不失为一种有益的补充性读物。

Original English language edition published by Manning Publications, USA. Copyright ©2013 by Manning Publications. Simplified Chinese-language edition copyright ©2015 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由Manning Publications 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2014-5768

图书在版编目（CIP）数据

驾驭文本：文本的发现、组织和处理/（美）英格索尔（Ingersoll, G. S.），（美）莫顿（Morton, T.S.），（美）法里斯（Farris, A.L.）著；王斌译.—北京：电子工业出版社，2015.7

书名原文：Taming text:how to find,organize,and manipulate it

ISBN978-7-121-25230-3

I. ①驾… II. ①英… ②莫… ③法… ④王… III. ①自然语言处理—研究 IV. ①TP391

中国版本图书馆CIP数据核字（2014）第302750号

策划编辑：符隆美

责任编辑：徐津平

印 刷：北京天宇星印刷厂

装 订：北京天宇星印刷厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编：100036

开 本：787×980 1/16 印张：21.25 字数：350千字

版 次：2015年7月第1版

印 次：2016年9月第2次印刷

定 价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

译者序

不知不觉，我进入信息内容处理这个领域已经有近20年了。这些年中，我的研究涉及机器翻译、Web搜索、跨语言检索、垃圾邮件过滤、问答、推荐、文本分类、聚类、情感分析等诸多技术或应用，也开发了多个原型以及实用系统。我十分高兴能够在这个有趣的领域不断地学习新技术，了解并开发新应用。与此同时，我也亲眼目睹了很多优秀的技术书籍不断涌现。完全出于兴趣爱好以及与大家分享的个人追求，我先后翻译了《信息检索导论》、《大数据：互联网大规模数据挖掘与分布式处理》、《机器学习实战》、《Mahout实战》等教材或技术书籍。现在，我又推荐给大家手边的这本《驾驭文本》。

文本处理是很多应用的基本技术，包括上面提到的搜索、推荐、问答应用都离不开文本处理。“驾驭”文本对于这些系统至关重要。然而，文本特别是自然语言文本本身的情况十分复杂，处理起来十分烦琐，难度很大。如何利用已有开源工具高效地“驾驭”文本是本书的目标。很显然，对于文本处理开发人员来说，这本书能够提供支撑。当然，由于自然语言文本固有的歧义性，文本处理技术特别是深层“理解”技术还远未成熟，研究人员还在不断努力，全方面真正“驾驭”文本是所有文本处理工作人员的终极梦想。

本书介绍了文本搜索、模糊字符串匹配、命名实体识别、文本聚类分类标注等多种文本处理关键技术，并通过融合上述技术构建了一个简单的事实型问答系统。所有的单项技术都有可供下载使用的数据集和相应的运行代码，读者可以下载这些

代码和数据进行尝试，以便能够更加深入地理解这些技术。

本书作者都是开源社区的重要贡献者，他们在文本处理领域具有丰富的开发经验。这些经验也都体现在本书的内容写作中。

感谢出版社和编辑部的辛勤工作，感谢实验室领导、同事以及译者家人对翻译本书的支持。

因本人各方面水平有限，现有译文中肯定存在许多不足。希望读者能够和我进行联系，以便能够不断改进。来信请联系wbxjj2008@gmail.com。

王 斌

2015年3月15日于中关村

序

在高质量文本处理需求持续指数级增长的年代，很难想象某个部门或业务不依赖某种类型的文本信息。迅速发展的Web经济也明显迅速加大了这种依赖性。与此同时，对高水平技术专家的需求也迅速增加。《驾驭文本》这本书就是应这种形势而出版的一本优秀的实用性书籍，它能够大量提供来自真实世界的经过实际验证的指导性案例。

Grant Ingersoll和Drew Farris是两位优秀的高水平软件工程师，和我一起工作过多年。而Tom Morton是在自然语言处理领域备受尊重的贡献者。他们仨联袂为我们奉献了一本实际课程的教材，该课程可以指导其他有志加入文本处理高级人才行列的技术人员，这些文本处理人才称为自然语言处理工程师。

本书采用学以致用的方法，为一个实际上十分复杂的过程褪去神秘的外衣。通过集中关注已有的工具、可实现的样例和已验证的代码，几位作者带领读者快速学习本来需要修一学期的NLP课程。

作为软件工程师，你已经具备基本能力能够跟进这些样例、代码和书中提到的开源工具，从而能够比预期更快地成为真正的专家，同时也能更快准备好面对来自实际世界的机会。

美国雪城大学信息研究学院院长 LIZ LIDDY

前言

生活中充满偶然瞬间，它们当中只有极少数会脱颖而出，就像那个确定我（Grant）职业生涯的瞬间一样。那是20世纪90年代末，当时我是一个年轻的软件开发人员，主要从事分布式电磁仿真的工作。有一天我看到一则广告，在纽约雪城（Syracuse）的一家小公司TextWise招聘一个开发职位。看完职位描述之后，我都没想过能获得这份工作，但是当时决定试试运气，就提交了一份简历。莫名其妙地，我获得了这份工作，于是开始了我的搜索和自然语言处理生涯。没想到这么多年以后，我仍然还在做搜索和自然语言处理，更没想到还会写一本这方面的书。

我那时候的第一个任务是开发一个跨语言信息检索（CLIR）系统，要求输入英语查询能够找到法语、西班牙语和日语文档，并将它们自动翻译成英语。回想起来，那个系统触及了我开始喜欢文本处理工作的所有难题：搜索、分类、信息抽取、机器翻译和所有那些奇怪的让每个学习文法的学生都疯狂的语言规则，等等。第一个项目之后，我后来又参与了多个搜索和NLP系统的开发工作，范围从基于规则的分类器到问答系统等。后来在2004年，NLP中心的一份新工作让我开始接触Apache Lucene，这个时代的开源搜索库（无论如何，至少目前还是）。后来我又参与开发一个CLIR系统，不过这次处理的是英语和阿拉伯语。因为需要一些Lucene功能来完成这项任务，我开始提交一些功能和错误的修正补丁。过了一段时间之后，我成为该社区的贡献者。从那之后，开源的“闸门”被轰然打开。我在开源领域涉入更深，并与Isabel Drost和Karl Wettin开始了Apache Mahout机器学习项目，并共同创立了一家

利用Apache Lucene和Solr进行搜索和文本分析的公司Lucid Imagination。

转了一圈之后，我认为搜索和NLP属于计算机科学的定义范围，不论是数据结构还是算法都需要复杂的方法来解决问题。除此之外，还有处理用户生成的大规模Web和社交内容的扩展性需求，这构成你的开发者之梦。这本书由工程师撰写给工程师，特别关注于使用现有、久经考验的开源库来解决文本处理中的疑难问题。个人认为目前这方面的市场还处于空白。我希望本书能够帮助解决当前工作中每天遇到的问题，也能激发你看到带来大量学习机会的文本世界。

GRANT INGERSOLL

我（Tom）在高二时就开始对人工智能感兴趣，本科毕业时选择去读自然语言处理方向的研究生。在宾夕法尼亚大学，我学习了大量文本处理、机器学习、算法和数据结构知识。我也有机会和自然语言处理领域最杰出的一些人共事并从他们身上学到很多东西。

在研究生阶段的课程中，我参加了多个NLP系统的开发工作，并参加了大量DARPA资助的有关共指、摘要和问答的评测。在这些工作中，我熟悉了Lucene和更大的开源运动。我也注意到能够提供高效端对端处理的开源文本处理软件还有较大欠缺。于是在我硕士论文的基础上，我为OpenNLP项目提供了大量贡献代码，并在之后的美国教育测试服务中心（Educational Testing Services）开发自动作文和短答案评分系统时继续学习NLP系统的一些知识。

在开源社区工作教会我很多与其他人一起工作的方法，也使我成为一名更优秀的软件工程师。现在，我在Comcast Cororation工作，与多个软件工程师团队一起使用本书中介绍的工具和技术。我希望本书能够在研究人员的艰难工作（这些工作就像我在研究生阶段学到的那样）与以使用文本处理来解决实际问题为目标的软件工程师之间架起桥梁。

THOMAS MORTON

和Grant一样，我是20世纪90年代中期由Elizabeth Liddy博士、Woojin Paik以及其他一些在TextWise进行研究的人员引入信息检索和自然语言处理领域的。我在完成雪城大学信息研究学院的硕士工作时和这个团队一起工作。那时，TextWise正处于从研

究组转型为创业公司的阶段，主要基于文本处理研究的成果开发商业应用。我在那个公司待了很多年，其间不断地学习和发现新的东西，并与一些优秀的同事一起共事，他们从各个角度来应对“教机器理解语言”这个挑战。

个人而言，我一开始是从软件开发人员的角度切入文本分析这个主题的。我有机会同优秀的研究人员一起工作，将他们的思想从实验转化为功能原型及大规模可扩展的系统。在此过程中，我有机会从事大量现在被称为“数据科学”的工作，发掘出对探索和理解大规模数据以及对它们进行学习的工具和技术的深深热爱。

怎样夸大开源软件对我职业的巨大影响都毫不为过。作为研究的伴随品，可用的开源代码为学习文本分析的新技术和方法以及软件开发提供了一条十分高效的途径。在这里我对所有尽力将知识和经验共享给那些有热情参加学习者的人表示敬意。我特别要感谢Apache软件基金会的那些好伙计们，他们为开源软件、人、处理过程和支持的社区贡献出一个不断成长的生机勃勃的生态系统。

本书中的工具和技术深深扎根于开源软件社区。Lucene、Solr、Mahout和OpenNLP都处于Apache这顶大伞之下。本书只介绍这些工具能实现的一些表面功能。我们的目标是提供对文本处理核心概念的理解，并为本领域的未来探索打下坚实的基础。

祝大家编程愉快！

DREW FARRIS

致 谢

本书经历很长时间完成，倾注了很多人的心血，这里要对他们表示诚挚谢意。

- 感谢Apache Solr、Lucene、Mahout、OpenNLP和其他本书中介绍的工具的用户和开发者
- 感谢Manning出版社，特别是和我们一直密切合作的Douglas Pundick、Karen Tegtmeier和MarjanBace
- 感谢本书的开发编辑Jeff Bleiel，感谢他在我们疯狂时间表的情况下仍然推进写作过程，感谢他一直以来的优秀反馈，也感谢他将我们这些开发人员转变为作者
- 感谢本书的评阅人，他们提出的问题、评论及批评提高了本书的质量。他们是：Adam Tacy、Amos Bannister、Clint Howarth、CostantinoCerbo、Dawid Weiss、Denis Kurilenko、Doug Warren、Frank Jania、Gann Bierner、James Hatheway、James Warren、Jason Rennie、Jeffrey Copeland、Josh Reed、Julien Nioche、Keith Kim、Manish Katyal、MargrietBruggeman、Massimo Perga、NikanderBruggeman、Philipp K. Janert、Rick Wagner、Robi Sen、SanchetDighe、SzymonChojnacki、Tim Potter、Vaijanath Rao和Jeff Goldschrafe
- 感谢在本书特定章节将专业知识贡献给大家的其他作者，他们是：J. Neal Richter、Manish Katyal、Rob Zinkov、SzymonChojnacki、Tim Potter和Vaijanath Rao

- 感谢Steven Rower，感谢他对本书所进行的全面的技术性评阅，也感谢他在TextWise、CNLP和部分Lucene项目时和我们一起共同度过美好时光
- 感谢Liz Liddy博士，感谢他将Drew和Grant引入到文本分析这个领域，感谢他带来的乐趣和机会，也感谢他为本书写序
- 感谢所有的MEAP读者，感谢他们的耐心和反馈
- 最重要的，要感谢我们的家人、朋友和同事，感谢他们的鼓励、精神支持以及对我们将正常的生活时间投入到本书写作的理解

Grant Ingersoll

感谢我在TextWise和CNLP的同事，他们教会了我太多文本分析的知识。感谢Urdahl让数学那么有趣，感谢Raymond女士让我成为一个更好的人和学生，感谢我的父母Floyd和Delores，感谢我的孩子Jackie和William，感谢我的妻子Robin，她忍受了我经常工作到深夜和写作占去的周末时光——谢谢你一直在那里支持我！

Tom Morton

感谢合作者的辛勤工作和团队合作，感谢我的妻子Thuy和女儿Chloe，感谢她们的耐心、支持和给予的自由时间；感谢我的家庭Mortons和Trans对我的鼓励；感谢我在宾夕法尼亚大学和Comcast的同事的支持和合作，特别要感谢Na-Rae Han、Jason Baldrige、Gann Bierner和Martha Palmer；感谢JörnKottmann为OpenNLP所付出的不懈努力。

Drew Farris

感谢Grant让我参与本书撰写和其他一些有趣的项目，感谢我过去和现在的同事，我从他们身上学到大量的东西并同他们共享文本分析、机器学习和开发优秀软件的乐趣；感谢我的妻子Kristin和孩子们Phoebe、Audrey和Owen，感谢他们对我挤时间写书和参与其他技术工作的耐心和支持；感谢我的大家庭，感谢他们的兴趣和鼓励，特别要感谢我的妈妈，虽然她已无法看到本书的完整版本。

关于本书

本书主要关注软件应用的构建，这些应用使用和处理书面文字的文本内容并从中掘取核心价值。尽管本书用较大篇幅介绍了有关搜索、自然语言处理和机器学习的主题，但是它并非一本有关这些主题的理论著作。我们尽量避免术语和复杂的数学公式，而集中关注当今软件工程师、架构师和从业人员为实现下一代智能文本处理应用时所需的一些概念和示例。本书也使用免费可用、高流行度的开源工具（如 Apache Solr、Mahout 和 OpenNLP）来提供书中一些真实世界中的实例的概念。

本书阅读对象

这书是否适合你？或许是。本书的目标读者是那些完全没有或没有太多搜索、自然语言处理和机器学习背景的软件从业人员。实际上，本书主要面对那些我们在很多公司看到的下面这种场景下的从业人员：某开发团队需要在一个新应用或在已有应用上增加搜索和其他功能，但是大部分开发人员并没有文本处理的经验。他们需要一个很好的入门材料来理解这些概念，同时又不会陷入那些不必要的内容之中。

很多情况下，我们提供容易访问的参考资源，比如维基百科和学术论文，因而本书可以作为读者需要对本领域进行深入探索的初始平台。此外，虽然本书大部分开源工具和样例都是基于 Java 的，但是本书的概念和思路也很容易移植到其他编程语言，因此 Ruby、Python 和其他语言的爱好者同样会对本书的内容满意。

尽管本书对需要实现教科书和学术书籍中的概念的学生有所帮助，但是其目标

读者很显然不是那些寻求相关系统中数学解释的用户和学术爱好者。

本书的目标读者也不是那些已经在其职业生涯中构建过很多文本处理应用的经验丰富的从业人员，尽管他们可能也会从本书开源包的使用中发现一些有趣的片段。不止一个有经验的从业人员告诉我们，本书可以加快本领域新人在文本处理应用开发中对思路和代码的理解。

最后，我们希望本书是一本面向现代程序员的最新指导书籍，也希望它成为文本处理应用编程职业道路之初所需要的指导书籍。

本书内容组织

第1章解释文本处理的重要性及其具有挑战性的原因。本章将预览一个基于事实的问答系统，以此来设定利用开源库驾驭文本的一个场景。

第2章介绍文本处理中的一些模块构建：切词、组块、分析及词性标注。之后考察利用Apache Tika开源项目从常见文件格式中抽取文本的过程。

第3章探讨搜索理论及向量空间模型的基本知识，重点介绍Apache Solr搜索服务器并给出利用它进行索引的方法。本章将学习如何对搜索性能（数量和质量）进行评估。

第4章考察基于前缀和 n 元组的模糊字符串匹配方法。我们考察两个字符串重叠度的计算方法：Jaccard和Jaro-Winkler距离，并解释如何利用Solr找到候选匹配并对它们进行排序。

第5章给出了命名实体识别背后的基本概念。我们将展示如何使用OpenNLP寻找命名实体并讨论OpenNLP的性能问题。我们还将介绍如何对OpenNLP进行定制从而在新领域中识别命名实体。

第6章主要介绍文本聚类。这一章会学习到常见文本聚类算法背后的基本概念，并且看到聚类如何提升文本应用的例子。我们也会介绍如何使用Apache Mahout来对整个文档集进行聚类，以及如何使用Carrot2对搜索结果进行聚类。

第7章讨论了分类、归类和标注背后的基本概念。我们会展示分类如何用于文本应用，并且介绍如何利用开源工具来构建、训练和评估分类器。我们还会使用Mahout中的朴素贝叶斯实现来构建文档分类器。

第8章综合前面7章学到的知识构建一个示例QA系统。这个简单的应用利用维基

百科作为知识库，并利用Solr作为基线系统。

第9章探讨搜索和NLP的下一步发展方向及语义、篇章和语用的角色。我们将介绍跨多种语言的搜索、内容中的情感探测，以及新兴的工具、应用和思想。

代码约定及下载

本书包含大量代码样例，所有源代码均采用等宽字体以区分普通文本。代码中的方法名称、类名称和其他元素也采用等宽字体表示。

在很多清单中，代码加以标注以指出重要概念，有时文本中也给出了项目编号来提供代码的额外信息。

本书的源代码样例与在线样例相当接近。但是为简洁起见，书中源码中去掉了像注释一样的内容，以保证代码能够方便地嵌入到文本中。

本书示例的源代码可以从出版社网站www.manning.com/TamingText下载。

作者在线

购买本书的读者能够免费访问Manning出版社管理的一个私有Web论坛，可以在这个论坛上发表对本书的评论、询问技术问题并从作者或其他用户那里得到帮助。你可以通过地址www.manning.com/TammingText访问和订阅该论坛。完成注册后，你可以了解如何访问论坛、该论坛所能提供的帮助及论坛的行为规范。

Manning出版社承诺为读者和作者提供一个进行深入对话的场所，但不对作者的参与程度做任何要求，他们对于该论坛的贡献出于自愿且没有任何报酬。我们建议读者尽量向作者提一些具有挑战性的问题，这样可以让他们保持兴趣！

本书在印期间，读者均可访问作者在线论坛，并查看之前的讨论。

关于封面

封面插图的标题是“Le Marchand”，是商人或店主的意思。该插图取自法国出版的Sylvain Maréchal的一个19世纪版本的四卷地域服饰习俗汇编。汇编中的每幅图都精心描画、手工上色。丰富多样的Maréchal作品生动地告诉我们，200年前的文化差异是如何之大，它将世界上的城镇和地域区分开来。人们彼此远离，说着不同的乡音和语言。不管是在街道或乡村，人们很容易通过服饰就能区分他们居住的位置、交易或购置的物品。

从那之后，服饰的密码逐渐改变，那时地域之间的丰富多样性也逐渐消失。现在很难区分来自不同洲的居民，更别说来不同城镇或地区的人了。或许我们以文化多样性为代价换来了更多样的个人生活，当然也是更丰富的快节奏的技术生活。

在一个很难分辨两本计算机书籍的年代，Manning出版社通过Maréchal的图画将我们带回过去，封面取材于200年前生活的多样性，借此颂扬计算机行业的创造力和首创精神。

目录

第1章 开始驾驭文本	1
1.1 驾驭文本重要的原因	2
1.2 预览：一个基于事实的问答系统	4
1.2.1 嗨，弗兰肯斯坦医生	5
1.3 理解文本很困难	8
1.4 驾驭的文本	11
1.5 文本及智能应用：搜索及其他	13
1.5.1 搜索和匹配	13
1.5.2 抽取信息	14
1.5.3 对信息分组	15
1.5.4 一个智能应用	15
1.6 小结	15
1.7 相关资源	16
第2章 驾驭文本的基础	17
2.1 语言基础知识	18
2.1.1 词语及其类别	19

2.1.2 短语及子句	20
2.1.3 词法	21
2.2 文本处理常见工具	23
2.2.1 字符串处理工具	23
2.2.2 词条及切词	23
2.2.3 词性标注	25
2.2.4 词干还原	27
2.2.5 句子检测	29
2.2.6 句法分析和文法	31
2.2.7 序列建模	33
2.3 从常见格式文件中抽取内容并做预处理	34
2.3.1 预处理的重要性	35
2.3.2 利用Apache Tika抽取内容	37
2.4 小结	39
2.5 相关资源	40
第3章 搜索	41
3.1 搜索和多面示例：Amazon.com	42
3.2 搜索概念入门	44
3.2.1 索引内容	45
3.2.2 用户输入	47
3.2.3 利用向量空间模型对文档排名	51
3.2.4 结果展示	54
3.3 Apache Solr搜索服务器介绍	57
3.3.1 首次运行Solr	58
3.3.2 理解Solr中的概念	59
3.4 利用Apache Solr对内容构建索引	63
3.4.1 使用XML构建索引	64
3.4.2 利用Solr和Apache Tika对内容进行抽取和索引	66

3.5 利用Apache Solr来搜索内容	69
3.5.1 Solr查询输入参数	71
3.5.2 抽取内容的多面展示	74
3.6 理解搜索性能因素	77
3.6.1 数量判定	77
3.6.2 判断数量	81
3.7 提高搜索性能	82
3.7.1 硬件改进	82
3.7.2 分析的改进	83
3.7.3 提高查询性能	85
3.7.4 其他评分模型	88
3.7.5 提升Solr性能的技术	89
3.8 其他搜索工具	91
3.9 小结	93
3.10 相关资源	93
第4章 模糊字符串匹配	94
4.1 模糊字符串匹配方法	96
4.1.1 字符重合度度量方法	96
4.1.2 编辑距离	99
4.1.3 n 元组编辑距离	102
4.2 寻找模糊匹配串	105
4.2.1 在Solr中使用前缀来匹配	105
4.2.2 利用trie树进行前缀匹配	106
4.2.3 使用 n 元组进行匹配	111
4.3 构建模糊串匹配应用	112
4.3.1 在搜索中加入提前输入功能	113
4.3.2 搜索中的查询拼写校正	117
4.3.3 记录匹配	122

4.4 小结	127
4.5 相关资源	128
第5章 命名实体识别	129
5.1 命名实体的识别方法	131
5.1.1 基于规则的实体识别	131
5.1.2 基于统计分类器的实体识别	132
5.2 基于OpenNLP的基本实体识别	133
5.2.1 利用OpenNLP寻找人名	134
5.2.2 OpenNLP识别的实体解读	136
5.2.3 基于概率过滤实体	137
5.3 利用OpenNLP进行深度命名实体识别	137
5.3.1 利用OpenNLP识别多种实体类型	138
5.3.2 OpenNLP识别实体的背后机理	141
5.4 OpenNLP的性能	143
5.4.1 结果的质量	144
5.4.2 运行性能	145
5.4.3 OpenNLP的内存使用	146
5.5 对新领域定制OpenNLP实体识别	147
5.5.1 训练模型的原因和方法	147
5.5.2 训练OpenNLP模型	148
5.5.3 改变建模输入	150
5.5.4 对实体建模的新方法	152
5.6 小结	154
5.7 进一步阅读材料	155
第6章 文本聚类	156
6.1 Google News中的文档聚类	157
6.2 聚类基础	158

6.2.1	三种聚类的文本类型	158
6.2.2	选择聚类算法	160
6.2.3	确定相似度	161
6.2.4	给聚类结果打标签	162
6.2.5	聚类结果的评估	163
6.3	搭建一个简单的聚类应用	165
6.4	利用Carrot ² 对搜索结果聚类	166
6.4.1	使用Carrot ² API	166
6.4.2	使用Carrot ² 对Solr的搜索结果聚类	168
6.5	利用Apache Mahout对文档集聚类	171
6.5.1	对聚类的数据进行预处理	172
6.5.2	K-means聚类	175
6.6	利用Apache Mahout进行主题建模	180
6.7	考察聚类性能	183
6.7.1	特征选择与特征约简	183
6.7.2	Carrot ² 的性能和质量	186
6.7.3	Mahout基准聚类算法	187
6.8	致谢	192
6.9	小结	192
6.10	参考文献	193
 第7章 分类及标注		195
7.1	分类及归类概述	197
7.2	分类过程	200
7.2.1	选择分类机制	201
7.2.2	识别文本分类中的特征	202
7.2.3	训练数据的重要性	203
7.2.4	评估分类器性能	206
7.2.5	将分类器部署到生产环境	208

7.3	利用Apache Lucene构建文档分类器	209
7.3.1	利用Lucene对文本进行分类	210
7.3.2	为MoreLikeThis分类器准备训练数据	212
7.3.3	训练MoreLikeThis分类器	214
7.3.4	利用MoreLikeThis分类器对文档进行分类	217
7.3.5	测试MoreLikeThis分类器	220
7.3.6	将MoreLikeThis投入生产环境	223
7.4	利用Apache Mahout训练朴素贝叶斯分类器	223
7.4.1	利用朴素贝叶斯算法进行文本分类	224
7.4.2	准备训练数据	225
7.4.3	留存测试数据	229
7.4.4	训练分类器	229
7.4.5	测试分类器	231
7.4.6	改进自举过程	232
7.4.7	将Mahout贝叶斯分类器集成到Solr	234
7.5	利用OpenNLP进行文档分类	238
7.5.1	回归模型及最大熵文档分类	239
7.5.2	为最大熵文档分类器准备训练数据	241
7.5.3	训练最大熵文档分类器	242
7.5.4	测试最大熵文档分类器	248
7.5.5	生产环境下的最大熵文档分类器	249
7.6	利用Apache Solr构建标签推荐系统	250
7.6.1	为标签推荐收集训练数据	253
7.6.2	准备训练数据	255
7.6.3	训练Solr标签推荐系统	256
7.6.4	构建推荐标签	258
7.6.5	对标签推荐系统进行评估	261
7.7	小结	263
7.8	参考文献	265

第8章 构建示例问答系统	266
8.1 问答系统基础知识	268
8.2 安装并运行QA代码	270
8.3 一个示例问答系统的架构	271
8.4 理解问题并产生答案	274
8.4.1 训练答案类型分类器	275
8.4.2 对查询进行组块分析	279
8.4.3 计算答案类型	280
8.4.4 生成查询	283
8.4.5 对候选段落排序	284
8.5 改进系统的步骤	287
8.6 本章小结	287
8.7 相关资源	288
第9章 未驾驭的文本：探索未来前沿	289
9.1 语义、篇章和语用：探索高级NLP	290
9.1.1 语义	291
9.1.2 篇章	292
9.1.3 语用	294
9.2 文档及文档集自动摘要	295
9.3 关系抽取	298
9.3.1 关系抽取方法综述	299
9.3.2 评估	302
9.3.3 关系抽取工具	303
9.4 识别重要内容和人物	303
9.4.1 全局重要性及权威度	304
9.4.2 个人重要性	305
9.4.3 与重要性相关的资源及位置	306

9.5 通过情感分析来探测情感	306
9.5.1 历史及综述	307
9.5.2 工具及数据需求	308
9.5.3 一个基本的极性算法	309
9.5.4 高级话题	311
9.5.5 用于情感分析的开源库	312
9.6 跨语言检索	313
9.7 本章小结	315
9.8 相关资源	315

第1章 开始驾驭文本

本章内容

- 理解文本处理的重要性
- 了解驾驭文本中的难点
- 准备好利用开源库来驾驭文本

如果你正在阅读本书，那么很有可能你是名程序员，或者至少在信息技术领域工作。当遇到电子邮件、即时通讯、Google、YouTube、Facebook、Twitter、博客以及大部分其他定义我们这个数字化时代的技术时，你可能应对起来相对得心应手。在你庆幸自己拥有非凡的技术能力的同时，不妨花点时间来设想一下你的用户：他们往往受困于所收到的大量邮件；他们在为组织生活中遇到的泛滥信息而努力抗争；他们或许不知道甚至不关心RSS或JSON，而对于搜索引擎、贝叶斯分类器或神经网络所知或许就更少；他们希望直接获得答案而不需要在结果页面中仔细筛选；他们希望邮件组织合理并且带有优先级，但是这些工作不需要他们自己花费什么时间。从根本上来说，你的用户需要一些使他们能够集中于自己生活和工作的工具，而不只是它们背后的技术。他们希望控制或驾驭“文本”这个不受控制的“野兽”。然而，驾驭文本到底意味着什么？我们将在本章的后面部分更多地讨论这一点，但是现在来说驾驭文本主要涉及如下三件事情。

- 找到包含信息需求的相关答案及支持内容的能力。
- 在很少甚至没有用户干预的情况下组织（标识、抽取、概括）和处理文本的

能力。

- 在输入量不断增长的情况下完成上述两个任务的能力。

这也就引出本书的主要目标，即给身为程序员的你一些工具和实际的建议来构建应用，以帮助人们更好地管理那些让他们陷入困境的信息“浪潮”。这本书的第二个目标是展示如何通过已有的高质量免费开源库和工具来实现上述应用。

在谈及本书后面那些更广的目标之前，我们先回来梳理文本处理涉及的一些要素及文本处理之所以困难的原因，并且也给出一些应用案例，这些是后续章节的动机所在。本章主要提供关于高精度文本处理为什么重要且具挑战性的背景知识。我们还会给出包含前面两个主要任务（即刚才提到的三个方面的前两个）的一个简单例子，同时也会给出本书最后建立的一个应用的预览。通过这两部分内容可以为后续章节奠定基础。该应用是一个基于事实的问答系统。通过这个例子，可以分析我们所在的信息世界的规模和形状，从而了解驾驭文本的一些动机。

1.1 驾驭文本重要的原因

这里只是为了好玩给出这样一个例子，假想我们在特定的某一整天没有阅读一个词。也就是说，有那么一天，没有阅读任何新闻、标志牌、网站甚至没有观看任何电视节目。想象一下你能否做到？几乎不太可能，除非你睡一整天。接下来花点时间来考虑一下阅读所需要的内容投入：长年累月的上学历程以及来自父母、老师和其他长辈的实际反馈，无数的拼写测验、语法课程和读书报告，更不要说接受大学教育所需花费的数十万美元。下面，我们从另一个层面退一步考虑一下你一天当中的内容阅读量。

一开始，花点时间考虑一下下面的问题。

- 今天你收到的电子邮件（工作邮件和个人邮件，包括垃圾邮件）有多少？
- 这些邮件中你阅读了多少？
- 有多少邮件你立即回复？一个小时内又回复多少？一天呢？一周呢？
- 如何寻找以前的邮件？
- 今天你阅读的博客有多少？
- 你访问的在线新闻网站有多少？

- 你和朋友或同事之间使用了即时通讯（IM）工具、Twitter或Facebook吗？
- 你在Google、Yahoo!和Bing上搜索了多少次？
- 你阅读了电脑上的哪些文档？它们是什么格式（Word、PDF或文本格式）？
- 你在本地（在本机或公司内网）搜索的频率有多高？
- 你以邮件、报告或其他形式产生多少内容？

最后，最重要的一个问题是，你做这些事花了多少时间？

如果你多少有点像典型的信息工作者的话，那么你很可能会与IDC（International Data Corporation，国际数据公司）2009年研究（Feldman 2009）中的发现有关：

每个工作者每周大概在邮件上平均花费13个小时……然而邮件不再是唯一的交流工具。社交网络、即时通讯、Yammer、Twitter、Facebook和LinkedIn等增加了新的交流通道，这些通道可能消耗信息工作者每天的集中生产时间。本年度用于搜索信息的时间是平均每周8.8小时，需要对每个工作者每年花费14209美元。而分析信息会耗费另外8.1小时，大概需要花费单位每年13078美元的投入。于是，对上述两个任务进一步自动化相对顺理成章。如果工作人员一天花费超过三分之一的时间来搜索信息，并花费另一个四分之一的时间来分析信息，那么该时间应该尽可能有成效。

再者，上述调查甚至没有说明同一个工作人员在私人时间内用于构建内容的时间。实际上，eMarketer估计互联网用户平均每周有18小时在线，而其他业余活动（如看电视）每周花费30小时，后者仍然是时间消耗大户。

无论是阅读邮件、搜索Google、阅读书籍还是登录Facebook，书面文字在我们的生活中无处不在。

刚才我们看到的是内容这张大图的个人部分，但是整体部分如何？按照IDC（2011年）的说法，2011年全世界共生成1.8ZB数字信息¹，而预计“2020年这个数字将扩大到50倍”。很自然地，由于我们无法预测下一个有可能产生超过预期内容的大趋势，通常上述预测值最后都被证明低于实际值。

虽然上述数据规模中很大一部分来源于数字信号数据、图像、音频和视频，当前使得这些数据可搜索的最好方法仍然是撰写分析报告、增加关键词标签或文本描

1 ZB=1024EB=1024²PB=1024³TB=1024⁴GB=1024⁵MB=1024⁶KB=1024⁷B≈10²¹B。——译者注

述，或者利用语音识别或人工隐藏字幕的方法对音频进行转录，以便可以将它们当做文本来处理。换句话说，不管增加了多少结构，对于我们而言它们仍然以文本的方式让大家共享和理解。正如你看到的那样，一方面，内容的巨大规模令人敬畏，另一方面，你将在下一节看到，即使在小规模数据上文本处理也是一个难题。与此同时，花时间考虑哪些理想的应用或工具能够帮助我们抑制那些将我们“吞没”的文本是值得的。对于很多应用而言，解决方案在于快速高效寻求问题答案的能力，而不只是给出一些可能的需要仔细检查的答案。此外，我们不必拐弯抹角地跨越层层障碍来提问，我们完全能够用自己的话或者声音来表达问题，而不是通过引用（quotation）、AND/OR运算符或者其他方式来表达自己，这些方式可能对机器很简单但是对人却很复杂。

尽管我们都知道我们并非生活在理想世界，驾驭文本中一个令人鼓舞的形式是问答系统，它可以处理自然语言（如英语）并返回实际答案，而不只是可能答案所在的页面。问答系统由于IBM参加Jeopardy!节目的Waston程序和苹果的Siri应用而为世人所知。在这本书中，我们会介绍构建这类系统的基础知识。为此，我们可以考虑一下这类系统的概貌，然后考查一段简单的代码，该代码可以从文本中抽取关键信息。本章最后，我们会深入了解该类系统以及其他语言应用的构建之所以困难的原因，并且看看后续章节如何为基于事实的问答系统及其他文本系统奠定基础。

1.2 预览：一个基于事实的问答系统

对于本书的目的而言，一个QA系统应该能够处理某个文档集，该文档集可能包含用户可能提的问题的答案。例如，维基百科或一些研究论文集合可以被用作答案寻找的信息源。换句话说，我们提出的QA系统基于包含答案文本的识别和分析来构建，而答案的寻找可以基于过去观察到的模式。该系统不能从多个数据源推测出答案。比如，如果对系统提问“Who is Bob's uncle?”，并且文档集中存在一篇文档包含“Bob's father is Ola. Ola's brother is Paul”，那么该系统无法从这两句话推理出Bob的uncle是Paul。但是如果有一句话直接给出“Bob's uncle is Paul.”，那么可以期望系统能够回答上述问题。这并不是说前面那个问题连尝试都不行，只是该问题的解决方法不在本书讨论范围之内。

构建前面提到的QA系统的一个简单流程如图1-1所示。

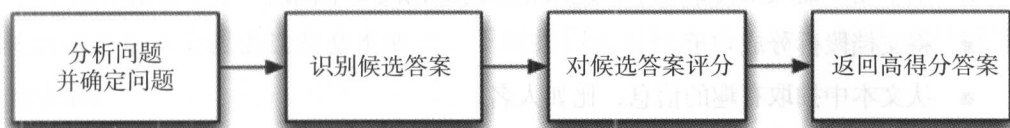


图1-1 对提交给QA系统的问题进行回答的一个简单流程

显而易见，上述简单流程隐藏了大量细节，也没有覆盖文档集的获取过程。但是，它强调了处理用户问题的几个关键部分。第一，分析用户问题并确定问题中关键点的能力通常需要类似词语识别之类的基本功能以及理解问题的答案类型的能力。例如，问题“Who is Bob's uncle?”的答案很可能是人，而问题“Where is Buffalo?”的答案则很可能是一个地名。第二，识别候选答案的需求通常会涉及可能包含答案的短语、句子或段落的快速查找过程，而不一定需要分析更大的文本单位。

对候选答案评分又意味着很多最基本的处理过程，比如词语分析以及对候选答案中是否真正包含问题答案必需内容（如必须提及某个人或某个地点）的更深理解。这些处理听起来比较容易，很多人也认为这些处理很容易，但是这并非理所当然。说完这些之后，我们看看下面的一个例子，该例子对大块文本进行处理，从中寻找段落并识别出像人名一样有意思的对象。

1.2.1 嗨，弗兰肯斯坦医生

前面讨论了问答系统以及文本处理的三个基本任务，接下来看看一些基本的文本处理过程。自然而然地，在这个简单系统中我们需要处理某个样本文本。为此，我们选择玛丽·雪莱的经典书籍《弗兰肯斯坦》²？为什么选择这本书？除了我们几个作者从文学角度喜欢这本书之外，这本书也碰巧是我们在Gutenberg项目（<http://www.gutenberg.org/>）中遇到的第一本书。它采用纯文本格式并且格式良好（你会发现在平时生活中这种格式很少见），另外一个好处是该书版权已经过期可以自由分发。我们的源码树下给出了该书的一个完整副本，你也可以从<http://www.gutenberg.org/cache/epub/84/pg84.txt> 直接下载这本书。

2 玛丽·雪莱的《弗兰肯斯坦》，又译作《科学怪人》，是西方文学中的第一部科学幻想小说。最初出版于1818年。后世有部份学者认为这部小说可视为恐怖小说或科幻小说的始祖。——译者注

现在我们拥有了一些可以处理的文本，下面做一些在文本应用中常常出现的任务。

- 基于用户输入来搜索文本，返回相关的文档段（本例是一个个段落）。
- 将文档段拆分成句子。
- 从文本中抽取有趣的信息，比如人名。

为完成上述任务，我们将利用两个Java库，一个是Apache Lucene，另一个是Apache OpenNLP，同时也会给出com.tamingtext.frankenstein.FrankensteinJava文件，该文件包含在本书中并且也可以从GitHub中获取（地址为：<http://www.github.com/tamingtext/book>）。源码的编译说明请参考<https://github.com/tamingtext/book/blob/master/README>。

驱动上述过程的高级代码如程序清单1-1所示。

清单1-1 Frankenstein驱动程序

```
Frankenstein frankenstein = new Frankenstein ();  
frankenstein.init ();  
frankenstein.index ();  
String query = null;  
while (true) {  
    query = getQuery ();  
    if (query != null) {  
        Results results = frankenstein.search (query);  
        frankenstein.examineResults (results);  
        displayResults (results);  
    } else {  
        break;  
    }  
}
```

使内容可搜索

提示用户输入查询

执行搜索

分析结果并显示有趣项

在上述程序中，首先对内容建立索引。构建索引是通过使用Lucene，使内容可搜索的过程。后面有关搜索的一章中我们会更详细地解释这一点。现在，可以将索引想象为一种在文本片段中查找单词出现位置的快速方法。接下来进入一段循环过程，在此过程中要求用户输入查询、执行搜索并处理返回的结果。基于本例的目的，可以将每个段落看成是搜索单位。这也意味着，当执行一次搜索之后，就可以清楚了解到到底是书中的哪个段落与查询匹配。

获得段落之后，可以切换用OpenNLP进行处理，它将每个段落拆分成句子，然后试图在句子中识别人名。我们不会考察每种方法的具体实施细节，因为本书后续的多个章节会覆盖这些基本概念。下面我们来运行程序，输入查询并观察返回的结果。

为运行代码，打开一个终端窗口（在命令行提示符下将当前目录改变到解压后源码所在的目录）并在UNIX/Mac系统下输入bin/Frankenstein.sh或者在windows系统下输入bin/frankenstein。你会看到如下结果。

```
Initializing Frankenstein
Indexing Frankenstein
Processed 7254 lines. Paragraphs: 722
```

```
Type your query. Hit Enter to process the query \
(the empty string will exit the program) :
>
```

此时，可以输入一个查询，比如“three months”。下面给出了部分结果的列表。注意，为了格式设置方便我们在很多地方插入了“...”符号。

```
>"three months"
Searching for: "three months"
Found 4 total hits.
```

```
-----
Match: [0] Paragraph: 418
Lines: 4249-4255
```

```
    "'Do you consider, ' said his companion to him, ...
```

```
    --- Sentences ---
```

```
    [0] "'Do you consider, ' said his companion to him, ...
```

```
    [1] I do not wish to take any unfair advantage, ...
```

```
-----
Match: [1] Paragraph: 583
Lines: 5796-5807
```

```
    The season of the assizes approached. ...
```

```
    --- Sentences ---
```

```
...    [2] Mr. Kirwin charged himself with every care ...
```

```
    >>>> Names
```

```
        Kirwin
```

```
...    [4] ... that I was on the Orkney Islands ...
```

```
    >>>> Locations
```

```

Orkney Islands
-----
Match: [2] Paragraph: 203
Lines: 2167-2186
    Six years had elapsed, passed in a dream but for one indelible trac
e, ...

----- Sentences -----
... [4] ... and represented Caroline Beaufort in an ...
    >>>> Names
        Caroline Beaufort
...    [7] While I was thus engaged, Ernest entered: ... "Welcome
, my dearest Victor, " said he. "Ah!
    >>>> Names
        Ah
    [8] I wish you had come three months ago, and then you would ha
ve found us all joyous and delighted.
    >>>> Dates
        three months ago
    [9] ... who seems sinking under his misfortune; and your pers
uasions will induce poor Elizabeth to cease her ...
    >>>> Names
        Elizabeth
    ...

```

上述输出结果给出了得分最高的那些段落，这些段落提到“three months”这个短语（总共4个段落），同时给出了段落中的一些示例句子，也给出了文本中的所有人名、日期和位置信息。在本例中，可以看到句子检测以及人名、地名和日期抽取的具体样例。如果目光敏锐一点的话也会发现上述简单系统中的明显错误。例如，系统认为Ah是人名，而Ernest却不是。该系统也没有将以“...said he. Ah!”结束的文本正确划分成不同的句子。这或许是因为我们的系统不知道如何正确处理感叹号或者文本中有一些稀奇古怪的格式。

现在我们暂时对上述错误的原因不做解释。如果进一步使用其他的查询，你可能会看到更多文本处理得好、坏甚至是丑陋的例子。上例也为下一节做好了准备，而下一节将会涉及文本处理的一些难点，该例同时也为本书中很多方法提供了动机。

1.3 理解文本很困难

假设Robin和Joe正在聊天，其中Joe说：“The bank on the left is solid, but the one

on the right is crumbling.”那么他们谈论的到底是什么？他们是在华尔街看着两个银行的办公室还是在密西西比河上泛舟并寻找停泊之处？如果是前者，那么这里的*solid*和*crumbling*可能指的是银行的资金状况，而如果是后者的话，那么这两个形容词指的是河岸的地面质量。现在，如果将上面谈话的两个人换成《汤姆·索亚历险记》³中Huck和Tom会怎样？那么你会相当确定他们是在谈论河岸而非银行，对吧？正如你看到的那样，上下文也相当重要。通常而言只有通过上下文的更多信息，再加上自己的经验，才能真正了解某段文本的真实含义。而刚才Joe所说的话还只涉及文本理解复杂性的表层。

如果是写作良好、具有连贯性的多个句子和段落，博识者能够无缝找到单词的意义并结合自己的经验和周边知识最终实现对内容和对话的理解。受过教育的成年人能够（或多或少）毫不费力地对句子进行剖析、识别关系并近乎瞬间推断出内容的含义。此外，正如上面Robin和Joe聊天的例子那样，当有某些内容在句子、段落或整篇文档中出现得不太合适或有缺失时，人们通常都能意识到这一点。人们也会在交谈中从他人那里获取信息输入，同时会立即调整语气和情绪来传达有关主题的想法，这些主题可以从天气到政治甚至某个指定击球员的角色。尽管我们常常认为这些技巧理所当然，但不要忘了它们是经过很多年的交谈、教育和反馈才逐渐调整成这样的，更不要说还来自于我们的先辈们传递下来的所有知识。

同时，计算机、信息检索（information retrieval, IR）及自然语言处理（natural language processing, NLP）领域仍然处于相对起步阶段。要接近人对内容的理解那样，计算机必须能够在不同层次上处理语言（要了解有关NLP各种因素的深度讨论，可以参考Liddy[2001]）。尽管完全理解对于计算机而言过于苛刻，但在面对海量文本及其出现的多样性时，即使是文本处理的基本任务也可能难以处理。

这就是为什么有一句谚语叫“the numbers don't lie”（数字不会撒谎）而不是“the text doesn't lie”（文本不会撒谎）。文本有不同样式和意义，即使是最聪明的人也会常常陷入理解的困境。建立一个文本处理的应用可能意味着要面对一系列技术和非技术的挑战。表1-2列出了文本应用面对的一些挑战，其中每一行的难度都比

3 《汤姆·索亚历险记》（*The Adventures of Tom Sawyer*）是一部美国著名的儿童文学作品，出版于1876年，作者为马克·吐温。Huck和Tom是其中的两个人物。——译者注

上一行有所增加。

表 1-1 文本处理会面对不同层次的挑战，从处理字符编码到根据周边世界所构成后的上下文环境推导出文本的含义

层 次	挑 战
字符	<ul style="list-style-type: none"> - 字符编码，比如 ASCII、Shift-JIS、BIG 5、Latin-1、UTF-8 和 UTF-16 等。 - 在不同应用中，大小写、标点符号、重音符号及数字的处理都不同
词和词素 ⁴	<ul style="list-style-type: none"> - 分词：将文本分成词。对于英语和其他利用空格隔开的语言来说十分容易，但是对于像中文和日文一样的语言来说就要难得多。 - 词性标注。 - 同义词识别，同义词对于搜索有用。 - 词干还原：将单词归约为其基本形式或者根形式。比如，words 进行词干还原之后得到 word。 - 省略语、首字母缩写及拼写对理解词语也相当重要
多词和句子	<ul style="list-style-type: none"> - 短语识别：比如 quick red fox、hockey legend Bobby Orr 和 big brown shoe 都是短语。 - 分析：将句子分解成主谓和其他关系常常会产生有关词和它们之间关系的有用信息。 - 句子边界识别在英语中很容易理解，但是仍然没有得到完美解决。 - 共指消解：以“Jason likes dogs, but he would never buy one”为例，he 指向 Joson。共指消解可能需要跨句进行。 - 单词常常有多个含义，利用单个或多个句子的上下文可以帮助选择正确的含义。该过程称为词义排歧，难以有效处理。 - 将单词定义及其关系组合起来确定句子的意义
多句和段落	在这个层次上，要对作者意图进行深入理解处理上变得更困难。摘要算法往往需要识别哪些句子更重要
文档	和段落类似，理解一篇文档往往需要超出实际文档本身的知识。作者常常期望作者具有一定的背景或阅读技巧。例如，如果读者从没有使用过计算机并且没有编过程，那么本书大部分意义不大。同样，大部分新闻报纸假定读者至少具有六级阅读水平
多文档和语料库	在这个层次上，人们想从一大堆文档中快速发现兴趣点或将相关文档聚在一起阅读它们的摘要。能够汇聚、组织事实与观点并发现它们关系的应用特别有用

在上述挑战之外，人的因素也在文本中扮演重要角色。不同文化背景、不同语言以及对相同作品的不同诠释可能会让最好的工程师也无所适从。对于整个文档集

4 一个词素（morpheme）是具有含义的一个很小的语言单位。前后缀就是词素的例子。

来说, 仅仅通过浏览一些样例文件就试图得到处理方法的做法往往导致很多问题。另一方面, 对大规模文档进行人工分析和标注又十分昂贵和耗时。但是可以放心的是, 能够找到帮助, 且文本可以被驾驭。

1.4 驾驭的文本

迄今为止, 你已经看到即将面临的一些挑战, 接下来振作精神来了解商业和开源社区(参考<http://www.opensource.org>)存在的一些处理工具, 这些工具能够对上述挑战进行一些处理。对你的文本处理旅途而言, 天大的好事就是事情在不断变化并且不断提高。感谢好的算法、更快的CPU、更便宜的内存、更低廉的存储器以及能够将多台计算机处理成单个虚拟CPU的工具, 十年前由于资源有限而无法处理的问题现在获得了积极的结果。尤其重要的是, 现在出现了高质量的开源工具, 它们能够为新思想和新应用提供基础支撑。

本书的写作目的就是将实际经验融入到开源工具中从而引导你进入自然语言处理和信息检索领域。至少在本书结束之时, 我们不可能覆盖NLP和IR的各个方面, 也不会讨论最前沿的研究结果。相反, 我们会集中关注那些可能对你驾驭文本最有影响的领域。

在关注诸如搜索、实体识别(寻找人物、位置等实体)、分组及标注、聚类及摘要提取之前, 我们可以构建应用来帮助用户快速方便地寻找和理解文本中的重要部分。

在谈及驾驭文本中所有兴奋点时, 尽管我们可能会讨厌有些扫兴者, 但需要提到很重要的一点, 在处理文本时并没有完美解决方案。很多时候, 两个人在浏览相同的结果时不仅对结果的正确性可能有不同的看法, 并且关于如何改进也非一目了然。此外, 对某个问题的修改可能会引起其他的问题。测试和分析与获得高质量结果具有相同的重要性。最终, 最好的系统将人置于交互循环当中并在可能时基于用户反馈来学习, 这一点就像聪明人从错误中和向同事学习一样。用户反馈也不一定是显式的。捕获点击信息、分析日志和其他用户行为能够了解用户如何使用系统从而提供有价值的反馈。考虑到这一点之后, 下面列出了一些提高你的应用和保持你头脑清醒的一般性技巧。

- 了解你的用户。他们是否对某种结构比较关心? 比如表格或列表? 或者只是

收集文档中的词语即可？他们是否愿意为了获得更好结果提供更多信息？或者就是越简单越好？他们是否愿意为获得更好的结果等待更长时间？或者只需要立即获得一个好的猜测性结果？

- 了解你的内容。使用什么格式（HTML、Microsoft Word、PDF、txt等）？哪些结构和特征很重要？文本当中是否包含一些行话、缩略语或者对于同样的内容是否有不同的表达方式？内容只集中于单个领域还是覆盖了很多主题？
- 测试、测试，再多一点测试。花时间（当然也别花过多时间）来权衡结果的质量和获得结果的代价。在权衡这门艺术上获得丰富经验。每项非平凡的文本应用都必须要在质量和扩展性两方面之间权衡。通过融合内容和用户的信息，往往就能够找到大部分时间满足大部分用户需求的质量和性能之间的最佳点。
- 有时最佳猜测能达到很好的效果。要寻求一些办法来给用户提提供置信度，这样用户就可以对返回结果进行合理决策。
- 在其他条件都一样的情况下，倾向于更简单的方法。并且，你会因为简单方法可以获得优质结果而感到惊讶。

另外，尽管处理非母语本身十分有趣，但本书主要集中于英语。值得放心的是，给定正确资源的话，本书很多方法都能直接用于其他语言。

此外还有一点需要指出，要解决的问题难度范围可能会覆盖相对简单到十分困难的范围，有时候困难到只好通过扔硬币来解决。例如，在英语和其他欧洲语言中，切词（tokenization，也称词条化）和词性标注算法已经工作得很好，而类似外语机器翻译、情感分析、文本推理等工具就难很多，它们在非受限环境下效果往往不是很好。

最后，文本处理非常像坐过山车。在一些“高峰”上，你的应用毫无错误，而在一些“低谷”它们却无任何正确之处。事实是，本书讨论或者广义NLP领域中的方法都不是问题的最终解决方案。还有最终机会可以让你继续深入挖掘或者留名。因此我们从现在开始，通过设定上下文，开始介绍从搜索到NLP的广阔世界，从而为后续章节的思想奠定基础。

1.5 文本及智能应用：搜索及其他

多年以来，搜索都是互联网应用中的王者。没有Google和Yahoo!之类的引擎，毫无疑问，互联网不会像今天一样无处不在。然而，随着开源搜索工具（如Apache的Solr和Lucene）的兴起，加上无数采集器及分布式处理技术的出现，至少在不需要巨型数据中心的小规模个人或公司搜索的情况下，搜索已经成为一般产品。与此同时，人们对搜索引擎的期望也不断提高。我们希望只输入一到两个关键词的情况下花更少的时间得到更好的结果。我们也希望自己发布的内容能够很容易地被检索和组织。

此外，公司面临着不断增值的巨大压力。每次像Google或Amazon之类的大玩家在信息访问上有所提高时，对其余玩家的门槛也会随之提升。5、10或者15年前，在发现数据时加入搜索功能已经足够，而现在搜索已经成为必要前提，一些改变游戏规则的玩家则使用了复杂的机器学习及深度统计分析算法来对海量数据进行搜索，而这么大规模的数据对人们过去来说，需要很多年才能理解。这就是智能应用的演化过程。越来越多的公司正在既定的领域，应用机器学习和深度文本分析方法为他们的应用带来更多的智能。

采用机器学习和NLP技术主要是基于大规模数据的实际应用需求，而不是基于机器理解“人类”或通过图灵测试（参考http://en.wikipedia.org/wiki/Turing_Test）这种宏伟的概念，虽然这些概念也值得一试。这些公司集中关注发现并抽取文本特征、汇聚用户点击、评分和评论等信息、对相似内容分组或生成摘要，最终将上述功能以用户更好发现和使用内容的方式展示出来，这将最终导致更多的交易、流量或其他任何目标。毕竟，如果发现不了你也就不会出钱购买所需信息，对吧？

那么如何开始做上面这么多事呢？一开始可以建立基线搜索系统（将在第3章讨论），然后考察使用日常生活中的概念来自动组织内容的办法。你不是用人工方式而是用机器去实现上述过程（当然有时需要帮助）。有了这些之后，接下来的几个小节会将搜索和内容组织的思想划分成三个不同领域，并给出一个能够把不同概念综合在一起的例子，该例子会在后续章节中进一步完善。

1.5.1 搜索和匹配

对于大部分驾驭文本的活动来说，搜索都是起点。这也包括我们提出的QA系

统，它需要依赖于搜索来对输入数据建立索引，并识别匹配用户问题的候选段落。即使必须要应用搜索之外的技术，你也可能使用搜索来发现文本或文档来进一步应用更先进的技术。

在第3章《搜索》中，我们会探讨如何获得可搜索、可索引的数据，以及如何基于查询返回文档。我们还会探讨搜索引擎如何对文档进行排序并利用该信息提高返回结果的质量。最后，我们会考察所谓多面搜索，即通过限制预定类别的结果来对搜索进行优化。这些主题都会以Apache Solr和Lucene构建的例子为基础来讨论。

对搜索技术熟悉之后，你会认识到搜索的结果受限于支持搜索的内容。如果用户查找的单词和短语不在索引中，那么就不能返回相关文档。在第4章《模糊字符串匹配》中，我们会考察基于查询拼写检查结果的查询推荐技术，也会介绍这些技术如何应用于数据库或记录链接任务来实现非一般数据库的联接。这些技术也常常不仅用于搜索的一部分，也用于更复杂的任务，比如在两个公司合并需要整合它们的客户名单时，需要识别两份用户档案所指的是不是同一用户。

1.5.2 抽取信息

尽管搜索能够帮你找到包含所需信息的文档，但通常需要识别更小单元的信息。比如，大规模文本集中识别专有名词的能力对于追踪犯罪行为十分有用，或者也对发现两个永远不会相遇的人物之间的关系极其有用。为实现上述功能需要使用小文本片段的识别和分类技术，这一小片文本通常只包含几个词。

在第2章《驾驭文本的基础》中，我们会介绍面向语言单位的词语（如名词短语）识别技术，这些技术可以用于识别文档或查询中可以组合在一起的单词。在第5章《命名实体识别》中，我们会考察如何识别专有名词和数字短语，并在不考虑其语言作用的情况下将它们分到不同的语义类别（如人名、地名、日期）中。对于第8章《构建示例问答系统》来说，这种能力相当基础。对上述两个任务来说，我们将使用OpenNLP并探索如何使用其现有模型，同时考虑构建更加吻合数据的新模型。和搜索和匹配不同的是，这些模型将基于已有的人工标注的内容来构建，然后使用统计机器学习方法来生成模型。

1.5.3 对信息分组

与抽取信息相反的是，可以通过对信息分组或添加标签为文本增加补充信息。比如，考虑一下，如果能够为邮件自动添加标签和优先级，那么就可以寻找所有类似的邮件，这样处理邮件就太方便了，即可你只需要关注那种需要立即注意的邮件，并在发送邮件时寻找支持内容。

一种常见的处理办法是将文本分类。事实上，信息抽取所用的技术也可以应用文本分组或将文档分类。然后，这些组往往可以用作搜索索引的多个面、补充关键词或者另外一种用户浏览信息的方法。即使在用户通过标注来提供类别的情况下，这些技术也可以推荐以前用过的标签。第7章《分类及标注》会给出文档分类的模型构建过程以及这些模型应用于新文档上提高用户体验的过程。

当已经驾驭文本、发现要找的信息并抽取所需信息时，你可能发现所得到的信息有点过多了。在第6章《文本聚类》中，我们会介绍如何对相似信息聚类的方法。这些技术可以用于识别冗余信息并在需要时去掉这些信息。这些技术可以用于对相似文档聚类，于是用户可以一次考察全部主题并一次了解多篇文档的关联性而无须阅读每篇文档。

1.5.4 一个智能应用

在本书倒数第2章《构建示例问答系统》中，我们会将前面章节所介绍的一系列方法综合在一起来构建一个智能应用。具体地，那章会构建一个基于事实的问答系统，它设计为寻找那些文本中的知识性问题。例如，给定正确的内容，你可以回答这样的问题“Who is the President of the United States?”。该系统会使用第3章《搜索》中的技术来识别可能包含问题答案的文本。而第5章《命名实体识别》中的技术会用于寻找那些往往是事实性问题答案的文本片段。第2章《驾驭文本的基础》和第7章《分类及标注》中的技术将用于分析所问问题并确定答案的类型。最后，你会利用第3章介绍的文档排序技术来对答案进行排名。

1.6 小结

驾驭文本是一项很大的任务，有时难以处理，另外由于不同语言、不同方言、

不同理解的存在而使得该任务更加复杂。文本可以是伟大作家笔下的优雅散文，也可以是什么风格或实质的烂文。不管格式如何，文本无处不在，并且需要人们和程序进行处理。幸运的是，不论是商业上还是开源社区，都有很多工具来帮助理解这一切。它们可能并非完美无缺，但是一直在变好。迄今为止，我们已经了解了一些文本重要性以及难以处理的原因。我们还了解了文本在智能Web中的重要角色，介绍了将要讨论的主题，并简单介绍了为构建一个简单的问答系统所需的相关技术。下一章我们闲话少说直接介绍文本分析的基础知识，并给出从当今的多种格式文件下抽取原始文本的基本技术。

1.7 相关资源

“Americans Spend Half of Their Spare Time Online.” 2007. Media-Screen LLC.<http://www.media-screen.com/press050707.html>.

Feldman, Susan. 2009. “Hidden Costs of Information Work: A Progress Report.” International Data Corporation.

Gantz, John F. and Reinsel, David. 2011. “Extracting Value from Chaos.” International Data Corporation. <http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf>.

Liddy, Elizabeth. 2001. “Natural Language Processing.” Encyclopedia of Library and Information Science, 2nd Ed. NY. Marcel Decker, Inc.

“Trends in Consumers’ Time Spent with Media.” 2010. eMarketer. <http://www.emarketer.com/Article.aspx?R=1008138>.

第2章 驾驭文本的基础

本章内容

- 理解文本处理中的切词、组块、分析及词性标注任务
- 使用Apache Tika开源项目从常见格式文件中抽取文本

很自然地，在开始学习文本驾驭核心过程之前，首先需要预热一下。一开始我们对高中英语进行一个简短的复习从而为后续内容奠定基础，我们会深入到切词、词干还原、词性、短语及子句等内容。后面你会看到，上述每一步工作都在结果的质量方面扮演着极其重要的角色。例如，看上去简单的切词可能会很难，尤其在像中文一样的语言中更是如此。即使在英文中，要正确处理标点符号也会使得切词十分困难。同样，由于语言中固有歧义的存在，文本中的词性标注和短语识别也相当困难。

下面将通过从现有的多种文件格式中抽取文本为例来讨论语言的基础知识。尽管很多书和论文提到内容抽取时基本都是打发打发，都假定已有纯文本可用，但是我们仍然觉得基于如下几条原因，需要考察内容抽取涉及的一些问题。

- 从一些专有文件格式中抽取文本通常十分困难。商业抽取工具也往往在抽取正确内容时失效。
- 事实上，你会花很多时间来了解多种文件格式和抽取工具，并看看是否正确处理。实际数据几乎不可能以简单的字符串方式出现。通常有很多奇怪的格式、随机的乱字符及其他问题，这些问题可能会让你挠破头皮。

- 后续处理只会和输入数据一样好。一个古老的谚语“垃圾进垃圾出”用在这里十分合适。

接下来，在复习英语知识和要抽取的内容之后，我们会介绍一些基础工具，这些工具会让你的应用和内容收藏更方便。言归正传，下面我们将介绍一些语言基础知识，包括如何识别单词、如何将它们组织成句子、名词短语甚至完整的分析树（有可能的话）一样的结构。

2.1 语言基础知识

你是否思念昔日初中时的美好时光？或许你想念中午英语课、图解句子、识别主谓关系并密切关注那些悬垂修饰语的那些时光。很好，你很幸运，这是因为文本分析的一部分就是回想中学英语及其他课程的基本知识。先把玩笑放在一边，下面几节会介绍分析文本必须要处理的一些常见问题，从而为构建相关应用打下基础。通过这种明确的基础构建，我们可以建立一个公共词汇表以便解释后面的概念，同时促使大家对语言的特点和功能及其在应用中如何使用进行思考。例如，到第8章《构建示例问答系统》时，你必须要能将原始字符串切分成一系列单词，并理解这些单词在句子中的角色（词性），以及它们之间如何通过短语或子句进行关联。给定这类信息之后，就可以接受“Who is Bob's uncle?”这样的问题，并仔细分析从而知道该问题的答案应该是一个专有名称（由多个标注为名词的单词组成），并且该答案必须和Bob、uncle处于同一个句子中，而且可能次序也相同。尽管这些东西对我们来说相当自然，但计算机必须要接受指令才能发现这些属性。尽管某些应用需要上述所有的构成部分，许多其他应用可能只需要其中的一个或两个。有些应用会明确给出这些构成部分在使用，而其他应用则不会。从长远来看，对于语言机理了解得越多，就能更好地评估任意文本分析系统内在的取舍之道。

在第一节中，我们会介绍词语的多种类别以及词语分组，并了解词语组合成句子的过程。我们简要介绍的内容属于语言中称为句法的领域，该介绍主要集中关注本书后面将会提到的主题。在第二节中，我们会考察单词的内部，这称为词法。尽管本书当中不会显式使用词法，但我们的基本介绍会帮助你理解我们给出的一些技术。最后，虽然我们研究的句法和词法可以应用于任意口语或自然语言，这里我们的例子将只限于英语。

2.1.1 词语及其类别

单词会分划分到多个词类或者词性中。这些类别包括名词、动词、形容词、限定词、介词和连词等。你可能已经在不同点见到过这些类别，但是可能并没有一次性见到所有这些类别或者了解它们的精确含义。基本熟悉这些概念对于后续章节十分重要，在那些章节中我们探讨的技术可能直接使用这些类别或者至少来源于这些类别。表2-1给出了基本的词类定义和示例信息。之后我们会探讨这些高级类别的额外细节。

表 2-1 常见词类的定义及示例

词 类	定义 ¹	示例（斜体部分）
形容词	命名属性的一个单词或短语，添加或语法上关联到某个待修饰名词，或者描述该名词	The <i>quick red</i> fox jumped over the <i>lazybrown</i> dogs
副词	修饰或限定形容词、动词或其他副词或词组的单词或短语，表达位置、时间、条件、态度、原因、程度等关系	The dogs <i>lazily</i> ran down the field afterthe fox
连词	连接两个词、短语或子句的词	The quick red fox <i>and</i> the silver coyotejumped over the lazy brown dogs
限定词	确定名词或名词组引用的类型的修饰词，如 <i>a</i> 、 <i>the</i> 、 <i>every</i>	<i>The</i> quick red fox jumped over <i>the</i> lazybrown dogs
名词	辨别人、地点或事物的词，或者给上述对象命名的词	The quick red <i>fox</i> jumped over the lazybrown <i>dogs</i>
介词	管辖名词或代词、常常出现在名词或代词之前的词，表示和其他词或子句中元素的某种关系	The quick red fox jumped <i>over</i> the lazybrown dogs
动词	描述动作、状态或发生的词，句子中谓语的主要构成部分，如 <i>hear</i> 、 <i>become</i> 和 <i>happen</i>	The quick red fox <i>jumped</i> over the lazybrown dogs

这些词类主要基于其句法用法而不是意义来定义，但是由于有些语义概念更可能用特定的句法结构表示，所以它们常常基于这些语义关联来定义。例如，名词常

¹ 这里所有的定义均来自《牛津美语大辞典》（New Oxford American Dictionary）第二版。

常定义为人、位置或某个事物，而动词定义为行为，但是我们也会使用像~~destrurction~~一样的名词，或者在“Judy is 12 years old.”中使用动词be，它们并不表现出典型的与名词和动词关联的语义关系。

这些高级类别往往还有更具体的子类，其中有一些在本书后面会用到。名词可以进一步分成普通名词、专有名词和代名词（即代词）。普通名词通常描述实体的类别（如town、ocean或person），它和专有名词区别很大，后者表示唯一实体并且通常首字母大写，比如London、John和Eiffel Tower等。代名词是引用其他实体的名词，这些实体通常在前面提到过。代名词包括he、she和it之类的词。其他词类也包括子类，甚至对于名词来说还有其他子类，但是对于这里的主题而言现有介绍已经足够。这些主题的其他信息可以通过搜索Web在很多很好的语法或语言类文献中获取，特别是维基百科，或者阅读文献（如The Chicago Manual of Style）或者收听Grammar Girl之类的播客（<http://grammar.quickanddirtytips.com/>）。

2.1.2 短语及子句

上一节给出的大部分词类都有对应的短语（多个词构成）结构。短语植根于某个特定类型的至少一个单词，但是也可以包含其他类型的单词和短语。例如，名词短语the happy girl包含一个限定词the、一个形容词happy和一个名词girl，其根为普通名词girl。这些短语的例子在表2-2中给出。

表 2-2 常见短语类型

短语类型	例子（斜体）	备 注
形容词	The <i>unusually red</i> fox jumped over the <i>exceptionally lazy</i> dogs	副词 <i>unusually</i> 和 <i>exceptionally</i> 分别修饰形容词 <i>red</i> 和 <i>lazy</i> ，从而形成形容词短语
副词	The dogs <i>almost always</i> ran down the field after the fox	副词 <i>almost</i> 修饰副词 <i>always</i> 从而形成副词短语
连词	The quick red fox <i>as well as</i> the silver coyote jumped over the lazy brown dogs	尽管这个例子有点例外，可以看到多个词构成的短语 <i>as well as</i> 在这里起到的功能像连词（如 <i>and</i> ）一样
名词	The <i>quick red</i> fox jumped over the <i>lazy brown</i> dogs	名词 <i>fox</i> 及其修饰语 <i>the</i> 、 <i>quick</i> 和 <i>red</i> 构成一个名词短语，同样名词 <i>dogs</i> 及其修饰语 <i>the</i> 、 <i>lazy</i> 和 <i>brown</i> 也构成一个名词短语

续表

短语类型	例子 (斜体)	备 注
介词	The quick red fox jumped <i>over the lazy brown dogs</i>	介词 <i>over</i> 和名词短语 <i>the lazy brown dogs</i> 构成一个介词短语修饰动词 <i>jumped</i>
动词	The quick red fox <i>jumped over the lazy brown dogs</i>	动词 <i>jumped</i> 及其修饰语介词短语 <i>over the lazy brown dogs</i> 构成一个动词短语

短语组合在一起可以构成子句，而子句是构建句子所需的最小单位。子句最少包含一个名词短语（主语）和一个动词短语（谓语），而谓语通常包含一个动词和另一个名词短语。短语 *The fox jumped the fence* 是一个子句，其中包含一个名词短语 *The fox*（主语）以及一个动词短语（*jumped the fence*），而后者由名词短语 *the fence*（宾语）和动词 *jumped* 构成。其他类型的短语可以作为可选项加入到句子中来表示其他关系。通过这些组成部分，你会看到任意句子可以分解成多个子句集合，而子句又可以分解成短语集合，短语又可以分解成特定词性的词语集合。确定词性、短语、子句以及它们之间关系的过程称为分析。如果你曾经用图示法表示过句子的话，那么你可能自己已经进行过上述句法分析。本章后面会考察能够执行上述任务的工具。

2.1.3 词法

词法是有关词语内部结构的研究。在大部分语言中，词由词素或词根和各种词缀（包括前缀和后缀）构成。这些词缀基于词的使用方式对词进行标记。在英语中，词主要基于后缀进行标记而标记规则基于词的词类。

普通名词和专有名词随数量发生变化而呈现两种不同的形式：单数和复数。单数名词由原形构成，而复数名词通常通过尾部添加 *s* 来标记。尽管普通名词和专有名词只包含两种形式，代名词会随数量、人称格和性别变化。尽管如此，代名词是一个封闭词类，总共只有 34 种不同形式，因此通常列举会更容易一些，而不是对它们的形态结构建模。由其他词类派生出的名词也包含上述变换的词缀。对于基于动词或形容词的名词而言，这包括表 2-3 和 2-4 中列出的后缀。

表 2-3 当名词基于动词时名词形态的例子

后 缀	例 子	动 词
-ation	nomination	nominate
-ee	appointee	appoint
-ure	closure	close
-al	refusal	refuse
-er	runner	run
-ment	advertisement	advertise

表 2-4 当名词基于形容词时名词形态的例子

后 缀	例 子	形 容 词
-dom	freedom	free
-hood	likelihood	likely
-ist	realist	real
-th	warmth	warm
-ness	happiness	happy

动词的形态体系更加复杂，它包括8种可能的动词形式，但所有规则动词只包含4种形式。一些不规则动词也可以在用作过去分词时采用en而不是一般的ed后缀。这些形式都在表2-5中给出。其余三种形式明显只适合于少数几个不规则动词，并且不使用普通后缀。

表 2-5 规则动词形态和不规则动词的普通过去分词后缀

后 缀	例 子	标 记 形 式
none	look	原形
-ing	looking	动名词
-s	looks	第三人称单数
-ed	looked	过去式
-en	taken	过去分词

形容词和副词可基于比较来标记，包括比较级和最高级。形容词原形tall添加后缀-er可以得到比较级taller，或者添加后缀-est得到最高级tallest。副词也可以通过同样变换得到比较级和最高级，比如near的比较级和最高级分别是nearer和nearest。

对词之间的关系和词本身结构有了基本了解之后，就可以马上使用软件进行处理，这些软件能够利用上述特性来驾驭文本。

2.2 文本处理常见工具

现在已经对语言的句法和语义有所了解，下面就看看一些数字文本处理工具，它们能够帮助识别上面和其他地方提到的一些重要信息。这些工具的某些部分每时每刻都在使用，而其他工具只是偶尔才使用。接下来的介绍会从字符串处这类基本工具开始，然后介绍更复杂的处理工具，如完整的句子分析。通常而言，最基本的工具每天都使用，而像完整语言分析器这种更复杂的工具只在特定应用使用。

2.2.1 字符串处理工具

字符串、字符数组及其他文本表示的处理库构成大部分文本处理程序的基础。大部分编程语言都包含基本的处理库，比如拼接、分割、子串搜索和一系列字符串比较方法。学习正则表达式库（如Java的java.util.regex包）会进一步提高你的能力（为全面了解正则表达式，可以参考Jeffrey Friedl的*Mastering Regular Expressions*这本书）。如果熟悉String、Character和StringBuilder类型，以及java.text包也十分有益。利用这些工具，你可以很容易得到文本的表层特性，比如某个词是否首字母大写？词有多长？是否包含数字和非字母字符等。此外，熟悉日期和数字的分析也十分有用。第4章中，我们会进一步深入了解字符串处理算法的细节。现在，我们只从语言上的动机来考虑文本的属性。

2.2.2 词条及切词

从文件中抽取内容的第一步基本都是将内容切分成小的可用的文本单位，该单位称为词条。词条往往代表单个词，但是正如你很快就要看到的那样，到底代表多大的文本单位取决于具体应用。第一种最常用的英语切词方法是基于空白字符（空

格或换行符），比如像下面的简单切词器`String [] result = input.split ("\\s+")`；就是如此。在这个简化的例子中，基于正则表达式`\s+`（注意Java中需要对`\`进行转义处理）输入String按照空格被分成一个String数组。尽管这种方法大部分情况下都有效，如果将代码运行于如下例句：

I can't believe that the Carolina Hurricanes won the 2005-2006 Stanley Cup.

就会得到表2-6所示的词条结果。其中，考虑最后一个单词Cup之后的句号是正确的做法，因为这可能会存在问题。

表 2-6 按照空格分割后的句子

I	can't	believe	that	the	Carolina	Hurricanes	won	the	2005-2006	Stanley	Cup.
---	-------	---------	------	-----	----------	------------	-----	-----	-----------	---------	------

尽管按照空白符来分割句子在一些情况下有效，但是大部分应用还要处理标点符号、缩略语、电子邮件地址、URL、数字等对象来获得更多好处。此外，不同的应用通常有不同的切词需求。例如，在Apache Solr/Lucene搜索库（将在第3章讨论）中，StandardTokenizer能够处理一些常见情况，如标点符号和首字母缩写词。实际上，去掉前面有关Hurricanes的句子尾部的句号之后，利用StandardTokenizer进行处理会得到结果如表2-7所示。

表 2-7 用 SolrStandardTokenizer 对句子进行分割后的结果

I	can't	believe	that	the	Carolina	Hurricanes	won	the	2005	2006	Stanley	Cup
---	-------	---------	------	-----	----------	------------	-----	-----	------	------	---------	-----

你可能会想：“我并不想去掉句号，我也不想让它附着在Cup后面。”这种想法是对的，按照应用的要求来做切词是思考的重点。由于像Lucene一样的搜索应用在大部分情况下都不需要句尾的句号，因此在上例中将句号去掉也没什么问题。

对于其他类型的应用而言，可能需要不同的切词方法。利用OpenNLP中的`english.Tokenizer`对上面那个句子进行处理会得到表2-8所示的结果。

表 2-8 用 OpenNLP 的 english.Tokenizer 对句子进行分割后的结果

I	ca	n't	believe	that	the	Carolina	Hurricanes	won	the	2005-2006	Stanley	Cup	.
---	----	-----	---------	------	-----	----------	------------	-----	-----	-----------	---------	-----	---

注意，这里的标点符号被保留且缩写`can't`被分开。这种情况下，OpenNLP将切词作为后续文法处理的前一步。对于这类应用来说，标点符号有助于判断子句边界，并且`can`和`not`具有不同的语法角色。

如果考虑OpenNLP中的另一类处理（命名实体识别，将在第5章讨论）的话，你会看到另一种切词方法。这里采用的策略是将字符串按照词条的类别进行分割，这些类别包括字母、数字、空格和其他字符。对于上面同一个句子，利用OpenNLP中的SimpleTokenizer进行处理得到的输出结果如下。

表 2-9 用 OpenNLP 的 SimpleTokenizer 对句子进行分割后的结果

I	can	't	believe	that	the	Carolina	Hurricanes	won	the	2005	-	2006	Stanley	Cup	.
---	-----	----	---------	------	-----	----------	------------	-----	-----	------	---	------	---------	-----	---

在上例中，你会看到时间范围被分割开。这样命名实体模块能对每个日期在时间范围内进行独立识别，这也是应用的需要。

正如你前面看到的那样，切词方法受目标任务的影响，满足任务需要的合适的文本单位受到文本处理类型的影响。幸运的是，大部分软件库会提供后续处理所需要的切词功能，或者提供方法来编写自己的切词器。

在词条层面进行处理的其他常见技术还包括以下几个方面。

- 大小写转换：在搜索中将所有词条转换成小写会有用。
- 停用词去除：过滤掉一些像*the*、*and*和*a*一样的常见词。这些常见词对那些不依赖于句子结构的的应用的价值微乎其微（注意，这里我们并没有说没有价值）。
- 扩展：增加同义词或者将首字母缩写和缩略语进行扩展，这样可以允许应用程序处理来自用户的输入的。
- 词性标注：对词条赋予词性。下一节将给出更多细节。
- 词干还原：将单词规约成词根或原形，例如将*dogs*转换成*dog*。更多细节在2.2.4节讨论。

下面将略过停用词去除、扩展和大小写转换等几方面内容，因为它们都是最基本的处理任务，通常只涉及简单的Map查找或String对象的基本方法调用。下两节中，我们会详细讨论词性标注和词干还原，然后从文本处理的词层面转向句子层面。

2.2.3 词性标注

识别单词的词性（part of speech, POS），比如确定单词是名词、动词还是形容词，往往可以用于提高后续处理的质量。例如，利用词性可以帮助确定文档中

重要的关键词（见Mihalcea [2004]，还有其他例子），或者可以用于辅助完成特定单词用法的搜索（比如*Will*是专有名词，而在“you will regret that”当中是情态动词）。在开源社区有很多现成的可训练的词性标注工具。其中一个OpenNLP的最大熵标注器，地址在<http://opennlp.apache.org/>。读者不要对于这里的短语“最大熵”太过担心，这只是一种利用统计来判断单词最有可能的词性的方法而已。OpenNLP英语词性标注器采用宾州树库项目（Penn Treebank Project, <http://www.cis.upenn.edu/~treebank>）中的词性标记来标注句子中词语的词性。这些标记中的很多也有一些相关标记用于标识单词的不同形式，比如过去式和现在式、单数和复数等。要浏览这些标记的完整列表，请参考http://repository.upenn.edu/cgi/viewcontent.cgi?article=1603&context=cis_reports。

表 2-10 常见词性的定义及示例

词 性	宾州树库中的标记名称	示 例
形容词、形容词最高级、形容词比较级	JJ, JJS, JJR	nice, nicest, nicer
副词、副词最高级、副词比较级	RB, RBR, RBS	early, earliest, earlier
限定词	DT	a/the
名词、复数、专有名词、专有名词复数	NN, NNS, NNP, NNPS	house, houses, London, Teamsters
人称代词、物主代词	PRP, PRP\$	he/she/it/himself, his/her/its
动词不定式、过去式、过去分词、现在第三人称单数、现在非第三人称、动名词或现在分词	VB, VBD, VBN, VBZ, VBP, VBG	be, was, been, is, am, being

现在你已经熟悉了我们后面将会遇到的一些词性标记，下面看看OpenNLP词性标注工具的工作流程。该工具使用了一个统计模型来工作，该模型通过考察许多已标注文档组成的语料库或文档集而建立，它包含用于计算词属于某个词性的概率的计算数据。幸运的是，你不必一定要构建这个模型（尽管你可以这样做），现在已经有一个模型可供使用。程序清单2-1给出了如何加载模型并运行词性标注工具的过程。

清单2-1 OpenNLP词性标注工具的例子

```
File posModelFile = new File (
    getModelDir (), "en-pos-maxent.bin");
FileInputStream posModelStream = new FileInputStream (posModelFile);
POSModel model = new POSModel (posModelStream);

POSTaggerME tagger = new POSTaggerME (model);
String[] words = SimpleTokenizer.INSTANCE.tokenize (
    "The quick, red fox jumped over the lazy, brown dogs.");
String[] result = tagger.tag (words);
for (int i=0; i<words.length; i++) {
    System.err.print (words[i] + "/" + result[i] + " ");
}
System.err.println ("n");
```

← 给出词性模型所在的路径

← 将句子切分成词

← 将切好词的句子传递给标注器

上述程序的运行结果如下：

```
The/DT quick/JJ , / , red/JJ fox/NN jumped/VBD over/IN the/DT lazy/JJ , / ,
brown/JJ dogs/NNS ./.
```

快速检查一下上述内容会发现结果还是比较合理的，其中*dogs*和*fox*是名词，*quick*、*red*、*lazy*和*brown*是形容词，而*jumped*是动词。迄今为止，这就是你需要知道的所有有关词性的知识，尽管在后续小节有时还会回顾这些知识。

2.2.4 词干还原

设想你的任务是阅读本国的所有报纸（希望你已经喝了咖啡！）来寻找有关bank的文章。于是，你得到一个搜索引擎并导入所有的报纸，然后就可以搜索bank、banks、banking、banker、banked之类的词。闪念之间，你可能会想：

“噢，我们可以希望能够只搜索 *bank* 就找到其所有的变形”，就像你意识到词干还原（或同义词，但是同义词属于另外一个主题）的威力一样。词干还原是将词归约成其词根或更简单形式的过程，归约的结果不一定是词本身。由于用户往往希望通过查找*bank*就能找到 *banking* 相关的文档，所以词干还原往往用于诸如搜索之类的应用当中。在大部分情况下，用户并不需要关键词的精确匹配结果，除非他们刻意要求这么做。

词干还原存在很多不同的方法，每种方法都有自己的设计目标。有些方法更大

刀阔斧也更激进一些，它们会将单词尽量归约为最小的词根，而有些方法则没有那么激进，而是优先考虑只做最基本的操作，比如从单词中去掉*s*或*ing*。它们之间的权衡体现在搜索这个例子当中，往往就是经典的匹配质量和数量之间的权衡。激进的词干还原方法会得到更多质量较低的结果，而非激进的方法会在失去某些有用结果的同时保留较高质量的结果。当有多种不同意义的词做词干还原处理得到同一词干时会遇到问题，此时原来的意义可能会丢失，而有些相关的词却不能归约成相同形式（参考Krovetz[1993]）。

如何选择某个词干还原器？什么时候使用它？同大部分NLP应用一样，这要视情况而定。运行测试、进行经验性判断并且多次反复试验可以最终确定上述问题的最佳实际答案。最后，一条最佳的建议就是通过用接口方式编码以便切换词干还原器。因此，可以一开始使用一个非激进的词干还原器然后从测试者或用户那儿收集反馈。如果用户认为该工具丢掉了一些单词变形，那么就可以使用更激进一些的词干还原方法。

现在已经了解了进行词干还原的原因，下面看看如何使用Martin Porter博士开发的Snowball这个词干还原的工具（地址为：<http://snowball.tartarus.org>）。除了友好的许可条款之外，Snowball词干还原器还支持多种方法并覆盖多种语言，目前包括的方法和覆盖的语言包括但并不限于下面这些。

- Porter and Porter2（称为EnglishStemmer）
- Lovins
- Spanish
- French
- Russian
- Swedish

清单2-2构造了一个英语词干还原器（有时称为Porter2），然后在一个词条数组上进行循环迭代。

清单2-2 使用Snowball英语词干还原器

```
EnglishStemmerenglish = new EnglishStemmer ();  
String[] test = {"bank", "banks", "banking", "banker", "banked",  
                 "bankers"};
```

构建待词干
还原的词条

```

定义期望
的结果
String[] gold = {"bank", "bank", "bank", "banker", "bank", "banker"};
for (inti = 0; i<test.length; i++) {
    english.setCurrent (test[i]) ;
    english.stem () ;                                ← 进行词干还原
    System.out.println ("English: " + english.getCurrent () ) ;
    assertTrue (english.getCurrent () + " is not equal to " + gold[i],
        english.getCurrent () .equals (gold[i]) == true) ;
}
通知
english
需要进行
词干
还原的词条

```

正如单元测试预期的那样，上述词干还原的结果是"bank"、"bank"、"bank"、"banker"、"bank"和"banker"。需要注意的是，按照上述英语词干还原器中的规则，*banker*及*bankers*都没有还原成*bank*。这不是上述英语词干还原程序出错，而只是上述算法运行的真实写照。尽管这对我们前面提到的阅读报纸的任务而言提高不是很大，但是仍然比将bank的各种变形都提交给搜索引擎要好。毫无疑问，在使用词干还原器的某种情况下，用户或者测试者可能会抱怨某个单词*X*没有被找到，或者该单词没有正确地进行词干还原。如果你真的认为要修正的话，那么最好的办法就是，使用一个受控词表来防止词被词干还原而不是试图修改词干还原算法本身（除非你控制了代码）。

2.2.5 句子检测

假设你想识别短语 *Super Bowl Champion Minnesota Vikings*² 在新闻中的所有位置，而你又遇到如下这段文字：

Last week the National Football League crowned a new Super Bowl Champion. Minnesota Vikings fans will take little solace in the fact that they lost to the eventual champion in the playoffs.

利用2.2.2节使用的 *StandardTokenizer* 对上述文字进行切词处理以后会产生如下词条。

```

... "a", "new", "Super", "Bowl", "Champion", "Minnesota", "Vikings", "fans",
"will", ...

```

如果你严格寻找词条*Super*、*Bowl*、*Champion*、*Minnesota* 和 *Vikings* 依次出现这

² 哎，我们可以做梦，行吗？

种情况，那么要做的就是短语匹配。但是我们知道，上述文字并不能严格匹配这个短语，原因就是 *Champion* 和 *Minnesota* 之间有一个句号。

句子边界的计算有助于减少上述短语匹配错误，同时也提供了一种识别词和短语以及句子和其他句子之间结构关系的方法。利用这些关系，就可以尝试发现文本中有意义的信息片段。Java中使用BreakIterator类来识别句子，但是通常还需要额外的编程来处理特例。例如，一个简单的利用BreakIterator的方法如下。

```
BreakIterator sentIterator =
    BreakIterator.getSentenceInstance (Locale.US) ;
String testString =
    "This is a sentence. It has fruits, vegetables, etc. " +
    "but does not have meat. Mr. Smith went to Washington.";
sentIterator.setText (testString) ;
int start = sentIterator.first () ;
int end = -1;
List<String> sentences = new ArrayList<String> () ;
while ((end = sentIterator.next ()) != BreakIterator.DONE) {
    String sentence = testString.substring (start, end) ;
    start = end;
    sentences.add (sentence) ;
    System.out.println ("Sentence: " + sentence) ;
}
```

运行上述BreakIterator示例后的输出结果如下。

```
Sentence: This is a sentence.
Sentence: It has fruits, vegetables, etc. but does not have meat.
Sentence: Mr.
Sentence: Smith went to Washington.
```

尽管BreakIterator能够正确处理嵌入的*etc.*，但它却不能正确处理*Mr.*。为改正这点，必须要在程序中加入代码来正确处理诸如省略号、引号以及其他可能的句尾符号。一个更好的选择是使用更鲁棒的句子检测程序，比如OpenNLP中的句子检测程序，最后的程序清单如下所示。

清单2-3 利用OpenNLP的句子检测

```
//... Setup the models
File modelFile = new File (modelDir, "en-sent.bin") ;
```

```

InputStreammodelStream = new FileInputStream (modelFile) ;
SentenceModel model = new SentenceModel (modelStream) ;
SentenceDetector detector =
    newSentenceDetectorME (model) ;
String testString =
    "This is a sentence. It has fruits, vegetables, " +
    " etc. but does not have meat. Mr. Smith went to Washington.";
String[] result = detector.sentDetect (testString) ;
for (inti = 0; i<result.length; i++) {
    System.out.println ("Sentence: " + result[i]) ;
}

```

利用 en-sent.bin
模型构建
SentenceDetector

调用检测过程

运行上述代码后得到如下正确的结果。

```

Sentence: This is a sentence.
Sentence: It has fruits, vegetables, etc. but does not have meat.
Sentence: Mr. Smith went to Washington.

```

2.2.6 句法分析和文法

NLP中的一个更具挑战性的任务是将句子分析成有意义的结构化集合或者关系树，它也是最有用的任务之一。例如，识别名词和动词短语及其关系有助于确定句子的主语以及与主体关联的行为。由于自然语言固有的歧义性，将句子分析成有用的结构十分困难。句法分析程序往往要在多种可能分析结果中做出抉择。例如，图2-1给出了下列句子的一颗分析树。

The Minnesota Twins, the 1991 World Series Champions, are currently in third place.

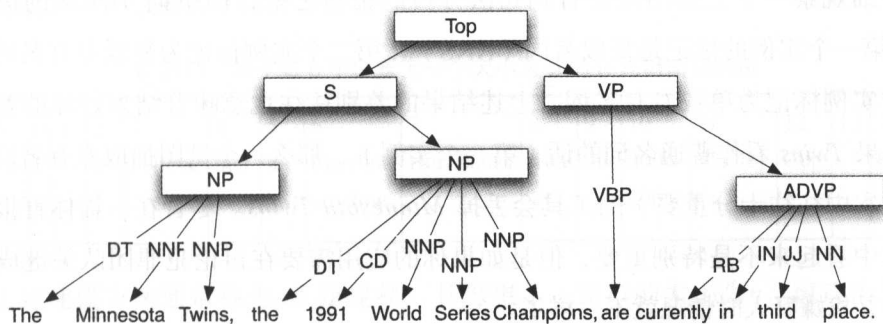


图2-1 利用OpenNLP分析器进行句子句法分析的一个例子

上述分析树通过使用OpenNLP Parser来构建，该分析器使用的是2.2.2节讨论的宾州树库项目中设计的语法。下列代码给出了分析的过程。

```
File parserFile = new File (modelDir, "en-parser-chunking.bin") ;
FileInputStream parserStream = new FileInputStream (parserFile) ;
ParserModel model = new ParserModel (parserStream) ;

Parser parser = ParserFactory.create (
    model,
    20, // beam size
    0.95) ; // advance percentage
Parse[] results = ParserTool.parseLine (
    "The Minnesota Twins , the 1991 World Series " +
    "Champions , are currently in third place .",
    parser, 3) ;
for (inti = 0; i<results.length; i++) {
    results[i].show () ;
}
```

在上述示例代码中，关键的一步是parseLine () 方法的调用，该方法输入待分析的句子返回可能的分析结果的最大数目。运行上述代码会产生如下输出结果（因显示需要做了截断处理），每行代表一种分析结果。

```
(TOP (S (NP (NP (DT The) (NNP Minnesota) (NNS Twins)) (, .) (NP (DT the)
...
(TOP (S (NP (NP (DT The) (NNP Minnesota) (NNPS Twins)) (, .) (NP (DT the)
...
(TOP (S (NP (NP (DT The) (NNP Minnesota) (NNP Twins)) (, .) (NP (DT the)
...
```

仔细观察一下上述结果会看到句法分析的精妙之处，即单词 *Twins* 的标注。*Twins* 第一个实例的标记是复数名词（NNS），第二个实例标记为复数专有名词，而第三个实例标记为单数专有名词。上述结果的差别往往也意味着结果好坏的差别，因为如果 *Twins* 看作普通名词的话（第一个实例），那么一个试图抽取专有名词短语（在文本中往往十分重要）的工具会丢掉 *Minnesota Twins*。尽管在一篇体育报道的上下文中看起来不是特别重要，但是如果你的应用需要在讨论犯罪团队关键成员的报道中寻找嫌疑人时情况就不一样了。

尽管上面给出的是一棵完整的分析树，并非总是需要完整（或称深层）的句

法分析。很多应用使用浅层分析时表现就很好。浅层分析识别句子的重要片段，比如名词和动词短语，但是并不一定需要将它们关联起来或者定义更细粒度的结构。例如，前面提到 *Minnesota Twins* 的例子中，某个浅层分析器可能只返回 *Minnesota Twins* 和 *1991 World Series Champions* 或某些类似的子句的变形。

句法分析是NLP领域中一个活跃的研究方向，研究成果十分丰富也很复杂，大部分内容都不在本书讨论范围之内。就本书的目的而言，句法分析用于诸如问答系统中帮助识别句子正确结构以便抽取答案这种地方。大部分情况下，句法分析都被看成是时机正当时使用的黑盒子，而不对其进行深入探究。

2.2.7 序列建模

迄今为止，上面讨论的方法提供了识别文本表层特征及语言属性的一种途径。下面我们考虑将文本看成词或字符序列来建模。一种常见的序列建模方法是考察序列中的每个元素和其前后出现的某些元素。到底考虑多少上下文随应用不同而不同，但是通常而言可以将此看成是上下文窗口，其中窗口的大小基于所考虑的上下文多少来设置。例如，使用字符序列对于在OCR（光学字符识别）扫描文本中搜索数据十分有用，此外也对短语匹配或者处理词间没有空格隔开的语言时有用。

例如，窗口大小为5，其中中间那个是待建模的字符，建模时会考虑它前面两个和后面两个元素。包含中间元素在内，我们需要考虑序列中的最多5个元素，考虑的元素个数就是窗口大小。当处理序列中的每个元素时，窗口可以看成在输入序列上“滑动”。为了保证窗口在句子所有位置上都有定义，通常边界上的单词会在句子前后附加多个开始或结束标记（参考表2-11）。

表 2-11 句子第 6 个位置上一个大小为 5 的 n 元组窗口示例

-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
bos	bos	I	ca	n't	believe	that	the	Carolina	Hurricanes	won	the	2005-	Stanley	Cup	.	eos	eos
						<-	-	window	-	->		2006					

上述建模方法通常称为 n 元组建模。其思想是，窗口的大小为 n ，窗口内的多个词组成 n 元组。表2-12给出了上面例句中不同大小的 n 元组样例。

表 2-12 不同大小的 n 元组

一元组	believe	Stanley	the	Carolina
二元组	believe, that	Stanley, Cup	t h e , Carolina	
三元组	believe, that, the	2005-2006, Stanley, Cup		
四元组	can't, believe, that, the	2005-2006, Stanley, Cup, .		
五元组	that, the, Carolina, Hurricanes, won			

相同的思想同样也可以应用字符建模，即每个字符看成是 n 元组窗口中的一个元素。这种基于字符 n 元组的方法将在第4章使用。

n 元组也很容易分解因此便于其自身进行统计建模和估计。在上述例子中，我们使用了5元组对单词 *Carolina* 的上下文建模。不太可能看到完全一样的单词序列，因此得到该单词序列的任何估计都很困难。但是，我们能够基于三元组 *the Carolina Hurricanes*，甚至二元组 *the, Carolina* 和 *Carolina, Hurricanes* 来估计一个概率。当上下文较大而与此同时存在容易估计的较短上下文时，这种回退估计的方法在文本处理当中十分普遍，该方法利用统计策略对文本中的歧义进行建模。

大部分类型的文本处理都会利用多种特征的组合，这些特征包括表层的字符串特性、语言单位和序列或 n 元组建模结果。例如，一个典型的词性标注器会使用句子和切词过程来确定标注元素，利用字符串处理来对前缀和后缀建模，并利用序列建模来得到待标注单词的前后单词。即使这些技术不能轻易获取文本的意义或语义，但它们却出奇有效。第5章就会使用上述三种处理的组合方法。

现在我们已经对文本处理的基础知识有所了解，包括字符串处理、语言处理及 n 元组建模，接下来我们考察开发文本应用时最常问并且首先遇到的问题：如何从常见格式（如HTML和Adobe PDF）文件中抽取文本？

2.3 从常见格式文件中抽取内容并做预处理

本节将展示如何从多种常见格式文件中抽取文本内容。我们会讨论预处理的重

要性并介绍一个开源框架，该框架能够从常见格式（如HTML和Adobe PDF）文件中抽取内容及元数据。

2.3.1 预处理的重要性

设想你在编写一个应用程序，通过输入关键词搜索本机的所有文件。为构建该应用，你必须要先让这些文件可以被搜索到。为了更好地了解要做的事情，看一下自己的硬盘，你马上就会意识到你有上千甚至上百万各种类型的文件。有些是Microsoft Word格式，有些是Adobe PDF格式，还有一些是像HTML和XML之类的文本格式，另外还有一些你无从下手的专有文件格式。作为一个聪明的程序员，你知道为了使这些文件类型有意义的话，就必须要将它们转换成一个公共格式，这样就只需要关心一种内部格式。这种将多个不同文件类型标准化为一个公共文本表示的过程称为预处理。预处理也意味着可以对文本加入或者修改信息以便被软件库更好地使用。例如，某些库期望输入中的句子已经被识别好。预处理基本上包含了在将输入数据传递给库或者期望应用之前所有必须要完成的步骤。作为预处理的一个主要示例，我们会考察从常见格式文件中抽取内容。

尽管有很多开源工具（很快就会讨论到）可以从不同类型文件中抽取文本，但是仍然值得投入资金购买文件转换软件库来插入到你的应用程序中。商业的文件转换器会向Microsoft和Adobe之类的公司支付许可费，这样就可以访问那些开源文件转换器无法访问的文档和程序库。此外，这些公司也会提供支持常规的维护和支持。这看上去很不公平，但是没人会说生活是公平的！但是在为商业工具付费之前，确保先获得一个评估版本并在自己的文档上进行测试。我们至少将一款著名的商业软件与一个开源Adobe PDF文件（PDF是一种很难抽取的文件格式）库进行过正面的比较，结果发现开源软件即使不比商业软件好，也至少和它相当。考虑到商业软件的价格，这种情况下显然要使用开源软件。你的情况可能与此有所不同，这依赖于你的内容本身。

由于这本书主要关注开源文本工具，如果不列举出可以用作预处理的开源工具，就有点不负责任。表2-13列出你可能会遇到的文件格式，同时也列出了一款或多款从该文件格式抽取内容的内置或开源库。考虑到这本书的性质，我们不可能在书中覆盖所有的文件格式，因此我们只关注那些在应用中很可能遇到的文件格式。不

不管你遇到的文件类型如何，也不管你是用开源库还是商业工具，大部分应用都希望有一个简单的基于文本的表示可供内部使用。这样就可以允许使用Java、Perl和其他现代编程语言中的基本字符串处理库。由于有很多不同的库，也有很多不同的内容抽取方法，因此也最好开发一个将文件格式映射为内容的框架或者使用一个已有的框架。

表 2-13 常见文件格式

文件格式	MIME 类型	开源库	备注
文本	plain/text	内置	
微软 Office (Word、PowerPoint、Excel)	application/msword, application/vnd.msexcel, 等等	1. Apache POI 2. Open Office 3. textmining.org	textmining.org 只能处理 Word
Adobe 便携文档格式 (PDF)	application/pdf	PDFBox	基于图片的 PDF 在没有使用 OCR 识别之前无法抽取文本
富文本格式 (RTF)	application/rtf	使用 RTFEditorKit 内置到 Java 中	
HTML	text/html	1. JTidy 2. CyberNeko 3. 很多其他工具	
XML	text/xml	很多 XML 库可用 (Apache Xerces 很流行)	大部分应用应该使用基于 SAX 而不是 DOM 的解析方法来避免构建重复数据结构
邮件	N/A	使用 Java 邮件 API 将邮件导出到文件中，或者使用 mstor	你的情况可能根据邮件服务器和客户端不同而有所不同
数据库	N/A	使用 JDBC、Hibernate 或其他工具，或者进行数据库导出	

谢天谢地，多个项目提供了预处理的框架。这些项目对表2-13中的很多库进行了包装。该方法能够让你对所有的库利用单个统一的接口来编写应用。我们使用这样

一个称为Apache Tika (<http://tika.apache.org/>) 的开源项目, 下面将对它进行介绍。

2.3.2 利用Apache Tika抽取内容

Tika是一个允许从许多不同源抽取内容的框架, 这些源包括Microsoft Word、Adobe PDF、文本及一些其他类型。除了对不同的抽取库进行包装之外, Tika也提供了MIME检测功能, 这样Tika就能自动检测内容的类型从而调用正确的库进行处理。

此外, 如果在Tika中没有找到你的格式, 不必担心, 通过搜索Web往往就可以找到能够处理该格式的某个插入(我们希望大家能对这些项目进行捐助回馈!)或独立使用的库或者应用。即使采用类似Tika这样的框架, 也建议你构建接口来包装抽取过程, 这是因为你很可能后来会增加某种文件格式, 而该文件格式又没有被Tika覆盖, 这需要一个十分清晰的方式来将它放入代码库中。

在架构层面, Tika的工作机理与SAX (Simple API for XML, 参考<http://www.saxproject.org/>) 解析器处理XML的机理很像。Tika从底层的内容格式(PDF、Word等)中抽取信息, 然后提供回调事件, 该事件可以被应用所处理。这种回调机制与SAX ContentHandler完全一样, 对于那些在其他项目上用过SAX的人来说应该十分直观。与Tika交互就像实例化一个Tika Parser类那么简单, 该类提供了单个方法:

```
void parse (InputStream stream, ContentHandler handler,  
            Metadata metadata, ParseContext parseContext)  
            throws IOException, SAXException, TikaException;
```

使用 parse 方法, 你只需要将内容作为一个 InputStream 传入, 而内容事件将被 ContentHandler 的应用实现所处理。内容中的元数据会跳到 Metadata 实例中, 其核心是一张哈希表。

在 Parser 这个层次, Tika带了多个可用的实现, 其中一个包含了Tika所支持的每个具体MIME类型, 而另一个 AutoDetectParser 能够自动识别内容的MIME类型。Tika也带了多个常见抽取场景下的 ContentHandler 实现, 这些场景包括比如只抽取文件的正文部分等等。

了解了Tika背后的基本原理及其基本接口之后, 看看如何利用该工具从多种不同

文件格式中发现和抽取文本。首先从一个抽取HTML文件最基本的情况开始，然后转向分析更复杂一点的PDF文件。

首先，假设你想解析某个相当简单的HTML文件。

```
<html>
  <head>
    <title>Best Pizza Joints in America</title>
  </head>
  <body>
    <p>The best pizza place in the US is
      <a href="http://antoniospizzas.com/">Antonio's Pizza</a>.
    </p>
    <p>It is located in Amherst, MA.</p>
  </body>
</html>
```

看到上述这个例子之后，你很可能想从中抽取标题、正文及可能的链接。Tika会使这一切都很容易，从清单2-4中就可以看到这一点。

清单2-4 利用Apache Tika从HTML中抽取文本

将抽取正文起始 标签之间文本的 Content Handler 了解 HTML 链接 的 Content Handler 抽取的元数 据保存之处	▶ InputStream input = new ByteArrayInputStream (html.getBytes (Charset.forName ("UTF-8"))); ▶ ContentHandler text = new BodyContentHandler (); ▶ LinkContentHandler links = new LinkContentHandler (); ContentHandler handler = new TeeContentHandler (links, text); ▶ Metadata metadata = new Metadata (); Parser parser = new HtmlParser (); ParseContext context = new ParseContext (); parser.parse (input, handler, metadata, context); System.out.println ("Title: " + metadata.get (Metadata.TITLE)); System.out.println ("Body: " + text.toString ()); System.out.println ("Links: " + links.getLinks ());	将多个 ContentHandler 包装成一个 输入的是 HTML， 因此构建其解析器 执行解析过程
--	--	--

上述代码运行于上面的HTML文件之后的结果如下。

```
Title: The Big Brown Shoe
Body: The best pizza place in the US is Antonio's Pizza.
It is located in Amherst, MA.

Links: [<a href="http://antoniospizzas.com/">Antonio's Pizza</a>]
```

上述解析HTML的代码涉及两个片段：一个是 ContentHandlers 的构造和Metadata

的存储,另一个是 Parser 的构建和执行,Parser 的作用如其名字所示,就是进行解析。在上述HTML例子中,可用HTMLParser来解析内容,但是大部分情况下你可能像使用Tika的内置 AutoDetectParser 类来解析一个PDF文件(如下例所示)。

清单2-5 利用AutoDetectParser来识别和抽取内容

```
InputStream input = new FileInputStream (
    new File ("src/test/resources/pdfBox-sample.pdf")); 构建 InputStream 来读入内容
```

```
ContentHandler textHandler = new BodyContentHandler ();
```

Metadata 对象
了作者,标题
之类的元数据

```
Metadata metadata = new Metadata ();
```

```
Parser parser = new AutoDetectParser ();
```

当调用 parser 时, AutoDetectParser 会自动估计文档的 MIME 类型。由于这里我们知道输入的是一个 PDF 文件,因此可以使用一个 PDFParser

```
ParseContext context = new ParseContext ();
```

```
parser.parse (input, textHandler, metadata, context); 执行解析过程
```

从 Metadata
列表中获取标题

```
System.out.println ("Title: " + metadata.get (Metadata.TITLE));
```

```
System.out.println ("Body: " + textHandler.toString ()); 从 ContentHandler 中打印出正文
```

在上述PDF文件的例子中,通过InputStream输入一个PDF文件,构造一个Tika中的ContentHandlers,同时建立一个Metadata对象来在合适位置上保存一些诸如作者和文档页数等等的附属信息以便在应用中使用。最后,构建一个Parser,对文档进行解析然后打印出文档的一些信息。正如你看到的那样,整个过程十分简单。

Tika能够很容易处理所有不同文件格式这个事实确实是一个好消息,更好的一个消息是:Tika已经通过一个称为Solr Content Extraction库(也称为Solr Cell)的贡献集成到Apache Solr中。在第3章中,我们会展示,将所有类型的文档送到Solr中建立索引搜索并不需要花费多大力气,这一切相当容易。此外,即使不使用Solr来搜索,也可以将它当成一个抽取服务器来用,这是因为它可以选择是否索引并返回文档的抽取结果。

2.4 小结

我们在本章考察了一些语言的基本知识,包括词法、语法、句法和语义,同时也介绍了一些它们的处理工具。之后,我们考察了一个常见的任务,即从文件中

抽取有用内容。尽管上述文本任务并不那么迷人，但它们往往十分必要。例如，很多人无视专用文件格式的抽取对文本分析的重要性，但是将内容抽取为可用文本往往费时费力还很难正确完成。同样，正如我们在本章介绍的那样，理解语言的基本知识对于本书其他内容的理解及理解领域的其他内容都有所帮助。考虑到这一点之后，可以通过考察很多文本分析系统中的基础部件，走出发现和组织内容的第一步，该基础部件就是搜索。

2.5 相关资源

Hull, David A. 1966. Stemming Algorithms: A Case Study for Detailed Evaluation. *Journal of the American Society of Information Science*, volume 47, number 1.

Krovetz, R. 1993 "Viewing Morphology as an Inference Process." *Proceedings of the Sixteenth Annual International (ACM) (SIGIR) Conference on Research and Development in Information Retrieval*.

Levenshtein, Vladimir I. 1996. "Binary codes capable of correcting deletions, insertions, and reversals." *Doklady Akademii Nauk SSSR*, 163(4): 845-848, 1965 (Russian). English translation in *Soviet Physics Doklady*, 10(8): 707-710, 1966.

Mihalcea, Rada, and Tarau, Paul. 2004, July. "TextRank: Bringing Order into Texts." In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP 2004)*, Barcelona, Spain.

"Parsing." Wikipedia. <http://en.wikipedia.org/wiki/Parsing>.

Winkler, William E., and Thibaudeau, Yves. 1991. "An Application of the Fellegi-Sunter Model of Record Linkage to the 1990 U.S. Decennial Census." *Statistical Research Report Series RR91/09*, U.S. Bureau of the Census, Washington, D.C.

第3章 搜索

本章内容

- 理解搜索理论及向量空间模型基础知识
- 搭建Apache Solr
- 使内容可搜索
- 为Solr构建查询
- 理解搜索性能

不论是作为应用的某个功能还是直接作为最终应用，搜索都无需太多介绍。搜索是我们生活中不可或缺的一部分，不管你是在互联网或本机上搜索信息，还是在Facebook上寻找朋友，或者在一段文本中查找关键词。对于开发者而言，搜索往往是大部分应用的关键功能，尤其那些用户需要对大规模文本进行筛选的数据驱动型应用更是如此。此外，搜索往往以事先解决方案的方式存在，比如桌面搜索工具Apple Spotlight，或者通过某种设备的方式提供，比如Google搜索设备（google search appliance，GSA）。

给定搜索普适性和事先解决方案可用性的情况下，一个很自然的问题就是，为什么需要使用某个开源方案来构建自己的搜索工具？原因至少包括下面几种。

- 灵活性：即使不能控制整个过程的全部环节，也能控制其中的大部分环节。
- 开发费用：即使购买商业解决方案，你仍然需要集成，这需要大量工作。
- 没有人比你更了解自己的内容：大部分密封包装的解决方法都可能对你的内

容进行某种不太合适的假设。

- 价格：这还用说吗？采用开源方案不需要许可费。

除了上述原因之外，开源搜索工具的质量也足以让人吃惊。与书中提到的任意一个其他工具相比，像Apache Lucene、Apache Solr之类的开源搜索工具不论在稳定性、扩展性还是第一时间的可实施性等方面都更胜一筹，它们已经应用到很多地方。本书当中，我们将基于Apache Solr来构建搜索项目，以便能够对内容进行快速的受控搜索。一开始我们将学习搜索引擎背后的一些基本搜索概念，包括Lucene和Solr的介绍。然后介绍如何搭建和配置Solr，之后对添加给Solr的内容进行索引和搜索。最后，我们会介绍一些提高Solr搜索性能的一般性技巧和技术。

首先，我们看看在线商店（如Amazon.com和eBay）中快速变成标准的两个搜索功能：搜索和多面。通过这个简单易懂的真实实例，我们可以为后续小节打好基础。我们可以开始规划如何将Amazon和其他网店的功能结合到自己的应用中去。本章最后，你不仅会理解搜索的基本知识，还将学会如何建立和运行一个真实的搜索引擎，并对如何加快搜索速度有更深的见解。

3.1 搜索和多面示例：Amazon.com

我们都遇到过这样的情况：你在某个购物网站上搜索，如果不浏览上百条结果的话就不能够确定所寻找商品输入的精确关键词。至少，在没有多面搜索功能的网站你会这样。例如，假设你要寻找一款全新的具有节电功能并且评价很好的50英寸液晶电视，于是你输入LCD TV，然后返回类似图3-1的结果。自然而然，第一页结果并非你所想要的结果，并且你也不期望这样，这是由于你输入了一条一般性的查询。因此，你开始缩小搜索的范围。在没有多面浏览方式的系统中，你通过加入关键词来实现这一点。比如，你的查询可能变成50 inch LCD TV或者Sony 50 inch LCD TV。但是由于你并不确切知道你所要的产品，而只是有一个想法，于是你开始考察Amazon提供的多面。面是指从搜索结果中导出的类别，它有助于缩小搜索范围。在Amazon这个例子中，如图3-1所示，这些“面”展示在网页左边的标题Category下，有一些像Electronics（4550）和Baby（3）之类的结果。每个面往往同时给出该面下的商品数量，这样就可以让用户决定是否感兴趣。

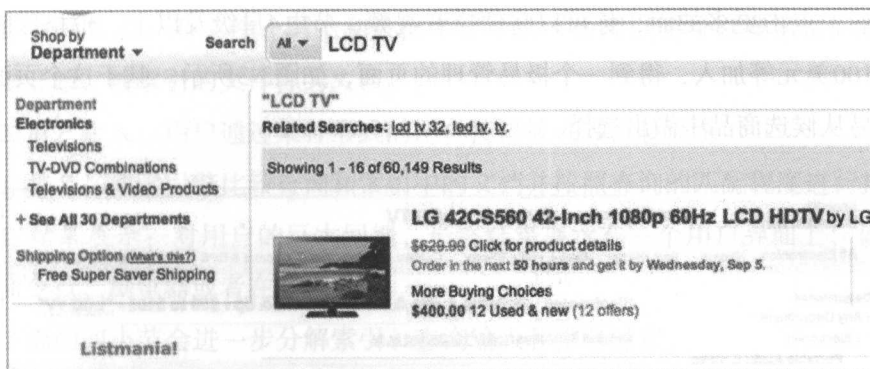


图3-1 查询“LCD TV”在Amazon.com上返回的结果摘要片段。2012年9月2日截取

继续上面那个查询样例，由于知道电视属于电子商品（electronic），因此你会点击Electronics这个面，得到图3-2所示的下一层“面”。注意这些“面”改变的过程会影响你的新选择。和前面一样，这里列出的类别既与搜索词项相关又属于上一次选中的“面”。并且，列出的每个类都能保证返回结果，这是因为真实数据确实存在结果中存在。



图3-2 搜索词项“LCD TV”在选择Electronics之后的面分布情况。2012年9月2日截取

最后，点击更多的面，你可以将便携音视频、节电4星级及以上、价格区间在50美元到100美元等加入，得到一个极易管理的页面，如图3-3所示。基于这个页面，你就很容易从候选商品中做出选择。



图3-3 通过多个面缩小搜索范围后“LCD TV”的搜索结果。2012年9月2日截取

上述Amazon的例子展示了多面搜索对于同时拥有结构化（元数据）和非结构化数据（原始文本）的网站来说是一个强大的工具，这些网站包括电子商务网站、图书馆及科技内容网站等等。对面进行计算需要浏览搜索结果集合相关的元数据，然后将它们组在一起计数（幸运的是，Solr为你做好了这些工作）。除了多面之外，很明显如果用户找不到某件商品，就不会购买或修理它，也不会仔细了解它的功能。因此，提高搜索能力并不是编程中某个深奥的训练过程，它会真正提高你的最低底线。为理解如何增加和提高搜索效果，下面先退一步看看一些基本的搜索概念，然后不断强化搜索过程。

3.2 搜索概念入门

在开始介绍实际搜索概念之前，我们先清除一下自己的记忆，忘掉所有你所知道的Web搜索的知识（至少这一会暂时忘记一下）。忘记Google，忘记Yahoo!和Bing，忘记PageRank（如果你熟悉的话）、数据中心和成千上个布满在互联网每个角落的CPU，这些CPU正为某个在线搜索引擎抓取每个可以纳入搜索的字节。我们拨开这些层层表面以便看到核心的搜索概念。按照最基本的定义，搜索可以表述成以下四部分。

1. 构建索引：对文件、网站和数据库记录进行处理以便可以搜索它们的内容。从这里开始，被索引的文件称为文档。
2. 用户输入：用户通过某种形式的用户接口输入其信息需求。
3. 排名：搜索引擎比较查询和索引中的文档并按照查询的匹配程度进行排名。
4. 结果展示：对用户的巨大回馈，最终结果展示在一个用户界面上，该界面可以在命令行、浏览器或者在手机上。

下面的四小节会进一步分解索引、查询输入和排名过程。

3.2.1 索引内容

不论你采用何种输入机制和排名算法，如果你没有很好理解文档集中内容的结构和类型，那么再多的数学公式得到的结果也不理解文档重点的搜索引擎输出的结果。例如，如果所有的文档都有一个标题属性并且你也知道标题匹配具有更大的信息量时，你可能会让搜索引擎给那些标题匹配的文档以额外的权重，这样就会提高其在结果中的排名。同样，如果你处理大量日期、数字、人名或者短语的话，你可能需要为正确索引内容做额外的工作。从反面来看，设想一个搜索引擎，它索引了在线新闻网站的所有HTML标签并且它不能区别这些标签和实际内容，那么可以想象该搜索引擎的质量可能会十分糟糕。显然，上面给出的只是一个虚构的示例，但是它强调了必须要开发一种迭代的方法来构建和提高应用使之更好地响应用户的需求。任何人要实现搜索的第一件事就是对要索引的内容有所了解。对内容调研应该同时覆盖文档的典型结构和实际内容。

对获得的内容有初步了解之后，使得这些内容可搜索的称为构建索引（indexing，也可简称索引）的过程就可以开始了。索引是搜索引擎使一个或多个文档可搜索的过程。为使某篇文档可搜索，索引过程必须要分析文档的内容。文档分析通常包括将文档切分为词条，之后一个可选的做法是对每个词条做一次或者多次改变来得到一个归一化的词条，最后的结果称为词项。从词条到词项进行的改变可能包括词干还原、全部小写或者直接剔除等做法。而到底如何选择改变方式取决于应用本身。有些应用可能不对词条进行任何改变，而有些却会进行大量修改。某些情况下，搜索引擎不会在分析的控制上提供太多方法。尽管缺乏控制一开始处理起来很容易，但当结果质量没有达到标准要求时仍然需要进行处理。表3-1给出一些将

词条转换为索引词项的常见方法。

表 3-1 常见的分析技术

技 术	描 述
切词	将字符串切分成多个索引词条的过程。对标点符号、数字和其他符号进行正确的、一致性的处理十分重要。例如，对 <i>microprocessor</i> 进行切分可能会输出多个词条（micro、processor 和 microprocessor），这样用户对这些变形的查询就更可能成功
全部小写处理	所有单词都转换成小写形式，这样就可以进行大小写非敏感的搜索
词干还原	去除单词的词缀等。参考第 1 章
去除停用词	去除常见词，如大部分文档中都会出现 the、and 和 a。这样做最早是为了节省索引空间，但是由于停用词有助于高级搜索，现代搜索引擎不再去除停用词
同义词扩展	对每个词条，通过同义词词典可以查到同义词，然后将其加入到索引中。这种做法通常基于查询词项而非索引词项进行，这是因为同义词表的更新可以在查询时动态考虑而不需要重建索引

从文档抽取出词项之后，通常它们存储在某个称为倒排索引的数据结构中，该结构为快速寻找到包含某个词项的文档进行了优化处理。当用户输入某个搜索词项，搜索引擎可以迅速找到所有包含该词项的文档。图3-4中给出了一个倒排索引的样例，它展示了索引中所有单词（有时称为词汇表）和其所在文档的链接关系。很多搜索引擎不仅仅是简单的词项到文档的索引结构，它们还记录了词项在文档中出现的位置信息。当判断两个或者多个词项互相邻近需要位置信息时，上述记录位置信息的做法就能使得短语查询或更高级查询的处理更加容易。

除了保存词项到文档的关系之外，索引过程往往还计算并保存词项在文档中的相对重要程度。这种重要程度的计算对于搜索引擎的能力来说至关重要，由此搜索引擎也实现了从简单布尔模型（判断词项在文档中是否存在）到可以按照文档相关度排序的排序模型的飞跃。如你所料，由于对文档排序可以让用户集中关注那些高相关度文档，因此在大规模信息的情况下，按照相关度对文档排序的能力是一个巨大的提高。此外，通过索引期间对这类信息尽可能的计算，搜索引擎能够在搜索时快速寻找文档并对它们进行排名。本章后面将对排名的实现过程做更多的介绍。现在，所需了解的索引的具体细节已经够用。接下来考察搜索者希

望对索引所做的事情。

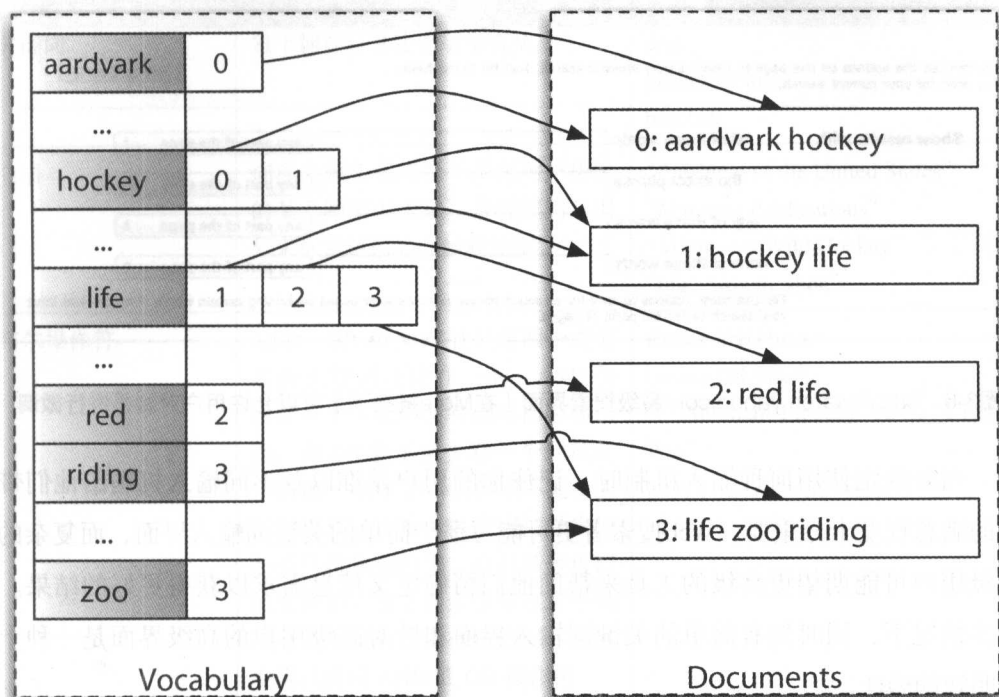


图3-4 将词项映射为所在文档的倒排索引数据结构，该结构便于搜索引擎快速查找查询词项。左边代表文档集中的一段词汇表，右边代表的是文档。倒排索引能够记录词项在文档中的位置

3.2.2 用户输入

搜索引擎一般通过一个用户界面（user interface, UI）来捕获用户的信息需求，该界面可以允许一个或多个输入，比如关键词、文档类型、语言、日期范围等等，然后按照相关度排名返回相关的文档列表。很多基于Web的搜索引擎依赖于一个简单的关键词方法，如图3-5所示，但是它们也提供更高级的查询输入机制，如图3-6给出的（部分）界面。

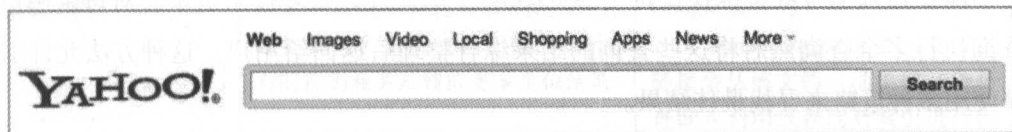


图3-5 <http://search.yahoo.com>给出了一个简单的用户搜索界面

图3-6 <http://search.yahoo.com>高级搜索界面（在More链接下）可以允许用户对结果进行微调

当要确定使用何种输入机制时，记住你的用户是谁以及不同输入机制给他们带来的满意程度十分重要。Web搜索者更可能习惯于简单的关键词输入界面，而复杂的高级用户可能期望更高级的工具来帮助他们精确定义信息需求以获得更好的结果。很多情况下，同时拥有简单的关键词输入界面和针对高级用户的高级界面是一种十分明智的做法。

多年以来，搜索引擎的查询功能不断增加，它们可以允许用户输入复杂查询，比如使用短语、通配符、正则表达式甚至自然语言的查询。此外，很多通用搜索引擎利用了像AND、OR、NOT一样的操作符及表示短语的双引号等符号来构建复杂的查询以缩小结果的范围。一些专门的实现甚至可能更进一步，提供直接处理内部数据结构、文件格式和查询需求的特定操作符。表3-2给出了搜索引擎当中常用的查询类型及操作符。搜索引擎可能会隐式构建任意一种查询类型，对此用户可能一无所知。例如，基于自然语言的搜索引擎（用户输入完整句子甚至段落作为查询）往往在查询处理时自动识别短语然后将短语查询提交给下一层引擎。同样，Google Canada的高级搜索界面提供了简单的文本框让用户构造复杂的短语和布尔查询，此时不需要输入引号或特殊操作符（参见图3-7）。此外，很多搜索引擎会对每条用户查询执行多个查询然后将这些查询的结果综合整理后返回给用户。这种方法允许引擎使用多种策略来寻找最佳结果。

表3-2 搜索中常见的查询类型和操作符

查询类型和操作符	描 述	例 子
关键词	每个词项分开在索引中查找	dog programming baseball
短语	词项必须相邻或者至少在用户指定的某个距离内出现。常常使用双引号来标识短语的开始和结束	“President of the United States” “Manning Publications” “Minnesota Wild Hockey” “big, brown, shoe”
布尔操作符	AND、OR 和 NOT 往往将两个或更多的关键词连接在一起。AND 表示在一个匹配中所有词项都必须出现，而 OR 表示至少有一个词项必须出现。NOT 表示该词项不能在匹配中出现。通常可以使用括号来控制操作符作用范围，并且括号可以嵌套使用。很多搜索引擎在多个词项查询上没有指定操作符时，实际在隐式使用 AND 或 OR 操作符	franks AND beans boxers OR briefs ((“Abraham Lincoln” AND “Civil War”) NOT (“Gettysburg Address”))
通配符和正则表达式	搜索词项可以包含通配符（? 和 *）或者成熟的正则表达式。这些查询类型常常比简单查询消耗更多的 CPU 资源	bank? ——寻找任意以 bank 开始并以任一字符结束的词，如 banks bank* ——寻找任意以 bank 开始以任意数目的字符结束的单词，如 banks、banker。 aa.*k ——所有 aa 开始 k 结束中间包含任意字符串的单词，如 aardvark
结构化查询	结构化查询依赖于待搜索的索引文档的结构。文档的常见结构包括标题、发布时间、作者、统一资源定位符（URL）、用户评分等	日期范围——所有时间在该范围内的文档 寻找某个具体作者 结果限制在某个互联网域下
相似文档	给定一篇或多篇已经找到的文档，寻找其他和这些文档相似的文档。有时称为相关反馈或更多类似结果	Google 曾经为大部分结果提供过一个 Similar Pages 链接。点击该链接会从该文档自动生成一条新查询并利用该查询对索引进行搜索

续表

查询类型和操作符	描 述	例 子
引导搜索	引导搜索或者称为多面浏览通过有效的类别为用户提供查询优化的建议，这种技术越来越流行	Amazon.com 使用多面浏览来允许搜索者通过限制价格区间、制造商或其他方面来进行搜索。每个面会有一个计数值代表该类别下的商品数量

Advanced Search

Find pages with...

all these words:

this exact word or phrase:

any of these words:

none of these words:

numbers ranging from:

图3-7 Google Canada的高级搜索界面，它能够自动构建复杂的短语和布尔查询，而不需要用户知道AND、OR、NOT之类的保留词或者对短语加引号

有些搜索引擎甚至走得更远，它们试图对查询的类型分类然后根据类型来选择不同的打分参数。例如，电子商务中的搜索查询通常属于如下两类之一：已知项和类别/关键词。当用户明确知道（或近似知道）搜索项的名称而只需知道其所在位置时就会进行已知项搜索。例如，查询 *Sony Bravia 53-inch LCD TV* 就属于已知项搜索，可能只有一到两个匹配项。类别搜索更普遍，它往往只涉及一些关键词：*televisions* 或 *piano music*。在已知项搜索的情况下，指定项目没有在前几项返回结果中出现就会认为是系统失败。而对类别搜索来说，由于输入的词项往往十分一般化，因此返回结果就会有一定的回旋余地。

向搜索引擎提交查询之后，为了实现与索引时类似的从词条到词项的转换，查

询词条会采用索引词条一样的方式进行分析处理。例如，如果词条在索引中做了词干还原处理，那么查询词条也要进行词干还原。很多搜索引擎也允许在查询时进行同义词扩展处理。同义词扩展是在用户定义的同义词词典中查找每个词条的技术。一旦匹配，那么新的代表同义词的词条会加入到词条列表中。比如，如果原始的查询词项是 *bank*，那么进行同义词扩展分析之后就可能会将 *finacial insitution* 和 *credit union* 加入到查询中，这个过程用户自己可能一无所知。同义词扩展也可以在索引时进行，但是这样做往往会导致索引规模的急剧增长并且在同义词表更新时需要重新索引内容。

到现在为止，我们已经讨论了搜索中涉及用户的基本问题，下面我们讨论向量空间模型和搜索的基本工作原理。对基础知识的了解有助于更好地在多种搜索方法之间做出权衡，并让你了解哪种方法满足你的需求。

3.2.3 利用向量空间模型对文档排名

尽管相对于其他讨论的主题来说，搜索（或者称信息检索）是一个相对成熟的领域，但这并不意味着已经找到了一种完美的寻找信息的方法。搜索任务有多种建模方法，每种方法都有它自己的优点。由于在我们选择的搜索库中使用了向量空间模型（vector space model, VSM），并且它也是给定查询对文档排名的最流行的方法之一，因此下面集中关注这种模型。如果想阅读更多的有关其他模型（包括3.7.4章的概率模型）的信息，可以参考Baeza-Yates与Ribeiro-Neto 合著的“Modern Information Retrieval”（Baeza-Yates 2011）或者Grossman 与Frieder 的“Information Retrieval: Algorithms and Heuristics（2nd Edition）”（Grossman 1998）。

向量空间模型内部机理的快速浏览

向量空间模型在1975年首次引入（Salton 1975）的一种代数模型，它将文档中的词项映射到 n 维线性空间。上面说起来就一句话，那么这到底是什么意思？设想你拥有一个受限语言生成的文档集合，整个语言只包括 *hockey* 或 *cycling* 两个词。假设以 *hockey* 为纵坐标、*cycling* 为横坐标将这些文档画到一个二维空间下。那么，一篇同时包含上述两个词的文档可以通过与坐标呈45度角的箭头（向量或词项向量）来表示（如图3-8所示）。

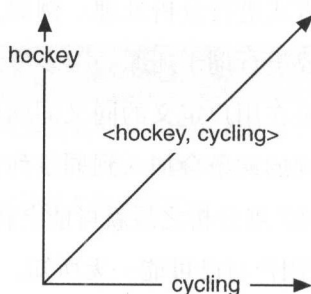


图3-8 一篇包含hockey和cycling两个词的文档的向量空间模型示例

尽管在二维空间下进行可视化展示十分容易，但是要将概念推广到多维空间却要困难很多。采用上述方法，可以将所有文档表示成 n 维线性空间下的向量。在这种表示中，文档集中的每个词代表空间的一维。例如，假设文档集中有两篇文档的内容如下。

- 文档1: The Carolina Hurricanes won the Stanley Cup
- 文档2: The Minnesota Twins won the World Series

于是，就可以通过对每个独立词语计数的方法，将上两篇文档映射到向量空间。例如，给词 *the* 赋予下标1, *carolina* 赋予2, *hurricanes* 赋予3, 直到所有文档中所有词都遍历一遍。你会看到当两篇文档包含相同词时，它们会在那一维有重合。图3-9给出了一个例子，将上述两篇文档映射到一个10维向量空间，其中词在文档中出现则意味着当前维保存一个为1的值。

Index	1	2	3	4	1	5	6
Doc 1:	The Carolina Hurricanes won the Stanley Cup						
Index	1	7	8	4	1	9	10
Doc 2:	The Minnesota Twins won the World Series						
Doc 1 Vector:	<1, 1, 1, 1, 1, 1, 0, 0, 0, 0>						
Doc 2 Vector:	<1, 0, 0, 1, 0, 0, 1, 1, 1, 1>						
Legend: A "1" in the vector means the word for that index number is present in the document; a "0" means it is not present in the document, e.g., "The" is in both documents, while "Carolina" is only in the first.							

图3-9 两篇文档表示成10维空间下的向量

图例：向量中的“1”意味着当前索引词项出现在文档中，而“0”意味着该词项没有出现在文档中，例如，“The”出现在两篇文档中，而“Carolina”只出现在第一篇文档中。

实际上，搜索引擎中的空间维度非常高（ n 往往大于100万）。基于存储和质量的原因，上面给出的那个简单的两篇文档的例子必须要做一些改变。对于存储而言，搜索引擎只存储词项的出现而不存储不出现的情况，于是就得到前面提到的倒排索引结构。由于大部分文档都只包含少部分词，这样就可以节省掉保存大量0所花的空间。而对质量而言，就不是简单地存在词时就存储1，大多数引擎会存储某类能够反映词项相对重要性的权重。用数学语言表达的话，上述处理实际是在进行向量缩放处理。通过这种方式，你可能就会开始明白，如果将查询中词项和包含这些词项的文档中的词项及其权重进行比较，就可以得到一个文档和查询相关的公式，后面我们很快就会提到。

尽管存在很多权重机制，最普遍的一种常常称为词项频率-逆文档频率（*term frequency-inverse document frequency*）模型，简称TF-IDF模型。TF-IDF模型的本质思想是，文档中出现频繁（TF），而在整个文档集中出现相对不频繁（IDF）的词比在大量文档中普遍存在的词更重要。将TF和IDF分别看成搜索的阴阳两面，它们之间相互折中。例如，在大部分英语文本中，*the*是常见词，这样其文档频率（DF）就很高因而IDF就很低，于是在对文档评分时它的出现就起不到什么贡献。这并不是说*the*和其他常见词（它们往往称作停用词）在搜索中没用，实际上它们会在短语匹配或者其他高级功能中发挥作用，当然这些内容超出了当前的讨论范围。另外一个极端就是，某个词在文档中出现多次（TF）但是在整个文档集中出现极少，那么它的价值很大，对于包含该词项的查询来说，它会对文档的排名起到重要作用。回到前面两篇文档的那个例子，*the*在第一篇和第二篇文档中均出现两次。而*the*总的文档频率为2，于是第一篇文档中*the*的权重为 $2/2=1$ 。类似地，*Carolina*只在第一篇文档中出现1次，因此在文档集上也出现1次，于是其权重为 $1/1=1$ 。对所有词项采用上述做法会得到完整的权重向量。文档1的权重向量如下。

```
<1, 1, 1, 1, 1, 1, 0, 0, 0, 0>
```

给定文档的向量空间表示，下一个合理问题就是如何在向量空间模型下对查

询和文档进行匹配。一开始，查询可以映射为相同空间下的向量。接下来，注意到将查询向量和文档向量的尾部对齐时，就会形成一个夹角。回忆一下高中的三角几何，对上述夹角求余弦会得到-1到1之间的一个值，这样就可以根据文档和查询的相似度将所有文档进行排名。很容易看到，如果两个向量的夹角为0度，那么就得到一个精确的匹配。由于0的余弦值为1，排名计算的结果也印证了上述理解。图3-10对上述概念进行了可视化展示，其中 Θ 表示文档向量 d_j 和查询向量 q 的夹角。图下面与每个向量关联的元组代表该向量中的权重值，这与前面介绍的两篇文档的例子很类似。最后，对文档集中的所有文档进行同样处理会产生返回给用户的排序文档列表。实际上，搜索引擎并不需要对所有文档打分，而只需要集中关注哪些包含一个或多个查询和文档共同词项的文档。此外，大部分搜索引擎在纯向量空间模型打分之外还补充了其他的特征和功能，比如文档长度、文档集中的平均文档长度等特征，还有能够将更多权重赋予一篇文档甚至一系列文档的算法等。

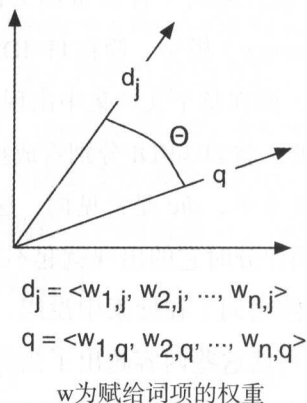


图3-10 用户查询 q 和文档集中第 j 篇文档的向量比较

自然而然，搜索引擎开发者也想出了计算这些得分的高效机制，于是利用向量空间模型可以在普通硬件上实现百万级甚至数十亿篇文档的亚秒级搜索。其技巧在于保证这些文档与用户的搜索相关。搜索速度和相关性都是3.6节的讨论主题。现在我们假设这些都已经提供，然后考察搜索结果展示的一些基本思想。

3.2.4 结果展示

如果你和我们的想法一样，那么结果展示通常是你心目中的最后一件事。顺序

抛出前10个结果并提供访问下一页结果的方法有什么不对？说实话，如果用户喜欢的话，这种方法没有任何不对。在这里，简单是崇高的设计目标，也是更多人需要记住的准则。然而，花费额外时间来确定结果展示的最佳方法能够大大增强用户交互的质量。但是要注意，即使看上去不错，精心设计的结果展示方式并不总能提供最有用的信息，所以要保证花费一定时间来考虑结果展示界面的可用性及其给用户带来的舒适程度。需要记住的有关结果展示的另外一些问题列举如下。

- 需要展示文档的哪些部分？如果标题存在的话，通常会给出标题。那么需要给出文档的摘要或者原始内容吗？
- 是否应该在结果中对查询中的关键词进行高亮显示？
- 如何处理重复或者近似重复的结果？
- 什么样的导航链接或菜单能够增强用户的体验？
- 如果用户不喜欢给出的结果，需要拓宽或者缩小搜索的范围又如何？

这些问题只能由你来回答并且依赖于你的具体应用。我们的建议是集中关注目标用户最想看到的结果，并且至少提供结果展示的功能。于是，你可以尝试一些可选方案给部分抽样用户使用，通过观察他们的反应来给他们选择替换方案的机会。

通过分析最常见的查询及用户行为，你不仅可以确定如何展示排序结果，还可以确定提供何种信息能够让用户快速找到所需结果。记住这一点之后，下面分析一些有助于组织结果的技术。

图3-11给出了Google中查询*Apple*的搜索结果。注意，在第一条结果中，Google添加了一些常见的*Apple*目标网页、股票信息和商店位置（甚至有地图）、相关人物等。这些信息下面是链接和相关搜索（图3-11没有显示这一部分）。

另外，在图3-11也会发现，为什么所有的结果中都不包含水果意义的Apple，比如澳洲青苹果和姬娜苹果？（这一页的下面有一个水果意义的搜索结果，但是Google将屏幕的主要部分都给了苹果公司的产品）很明显，由于Apple公司的流行度，它应该是排名最高的网站，但是如果你能根据结果之间的关联将它们分成多个组，你会怎么做？这就是为什么多面浏览及与其相近的聚类概念逐渐流行的原因（聚类将在第6章介绍）。利用分类或聚类技术，搜索结果可以根据文档属性分成多个组。在多面浏览的情况下，文档通常事先赋予了类别。而对结果聚类显示的话，聚类会基于返回结果文档的相似性动态进行。两种情况下，用户都可以修正结果或者将显示结

果限制在确保有效的类别当中。换句话说，通过选择其中一个类别，用户知道在该类别下存在相关结果。

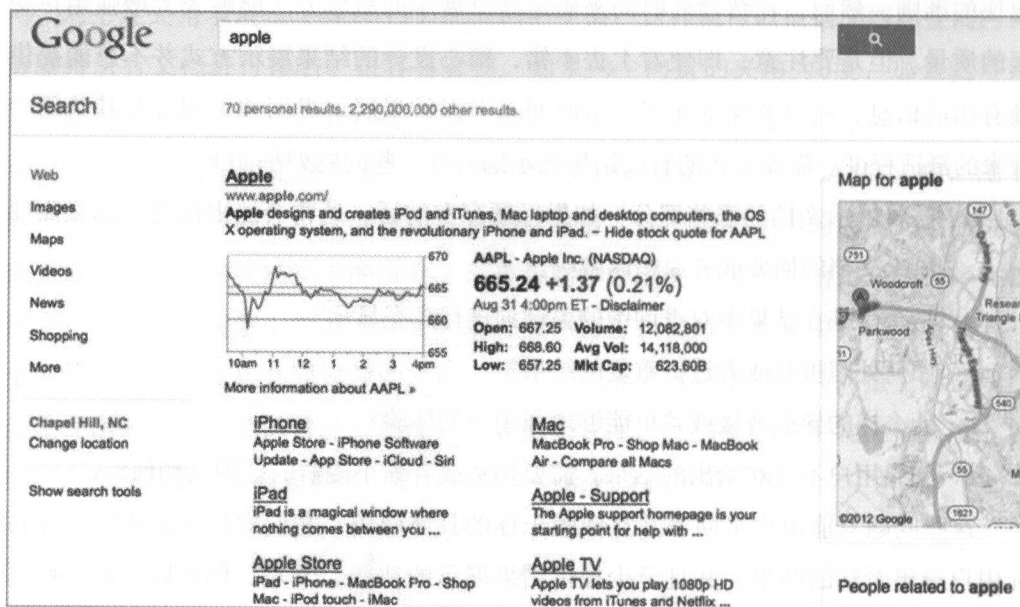


图3-11 Google在搜索结果显示时除简单排序列表之外还提供了更多选项

搜索结果聚类也能提高显示的结果。例如，Carrot搜索（<http://www.carrotsearch.com>）提供了一种对大量不同搜索引擎的返回结果进行聚类的机制（参考主页上的“Live Demo”）。图3-12给出了搜索图3-11中的同样查询Apple的结果。Carrot搜索的左边显示的是结果聚类的簇。利用这种显示方式，很容易基于特定的簇来缩小搜索的范围，比如可以如图3-12所示搜索苹果派而不是Apple公司。在第6章我们还会讨论如何利用Carrot来对你的搜索结果聚类。

在讨论Apache Solr之前，我们知道很多有关信息检索的好书，还有网站、兴趣组、社区和大量学术论文等，这些资料介绍了大量如何进行搜索的知识。因此，如果你在寻找某个特定类型的搜索引擎，或者只是想了解更多的信息，就可以从你喜欢的Web搜索引擎出发搜索*information retrieval*。另外，美国计算机学会（ACM）中有一个称为SIGIR的特别兴趣组，该兴趣组每年会举办一次学术年会，本领域最优秀最聪明的研究人员会在会议上与大家分享自己的进展。

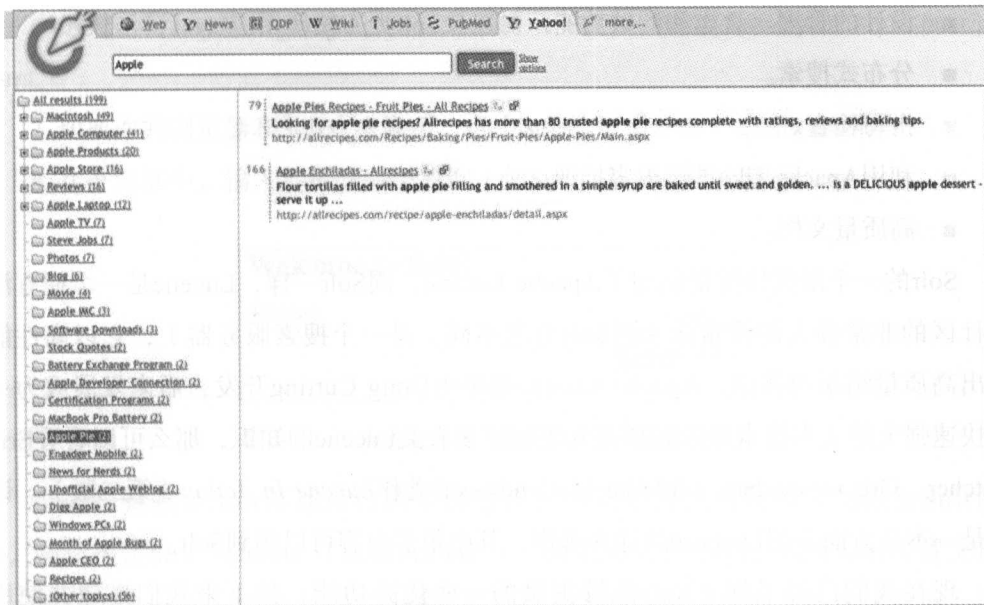


图3-12 对于有歧义的搜索词项，聚类是一种十分有用的结果展示方法，在这个例子中，Apple可能代表的是Apple公司或者一种水果，或者其他东西

现在的高级概念已经够了，是不是？接下来我们通过Apache Solr来考察如何在应用中嵌入一个实际搜索引擎。

3.3 Apache Solr搜索服务器介绍

Apache Solr（<http://lucene.apache.org/solr>）是一个基于Apache Lucene的线程安全、高性能的工业级搜索服务器。Solr最初构建于CNET，于2006年初捐赠给Apache软件基金会。从那之后，它增加了很多新功能，也存在大型活跃社区来讨论新功能的增加、错误修正和性能提高等问题。由于提供了如下的诸多关键特性，Solr已成为一个高品质搜索服务器。

- 提供了很简单的基于Http的索引和搜索协议，同时提供了Java、PHP、Ruby和其他语言的客户端。
- 提供了高级的缓存和复制机制，能够提升性能。
- 易配置。
- 提供多面浏览功能。
- 命中高亮显示。

- 设计的管理、日志和调试功能，可以让Solr更方便使用。
- 分布式搜索。
- 拼写检查。
- 利用Apache Tika进行内容抽取。
- 高质量文档。

Solr的一个最优特性是使用了Apache Lucene。同Solr一样，Lucene是一个拥有活跃社区的非常强大的搜索库（而Solr与之不同，是一个搜索服务器），它以高性能输出高质量结果而著称。Apache Lucene最早由Doug Cutting开发，后来发展成为一个快速强大的文本搜索库。如果需要了解更多有关Lucene的知识，那么可以看看Erik Hatcher、Otis Gospodneti和Mike McCandless的著作*Lucene In Action*（第二版）。该书是一本全方面介绍Lucene的优秀著作，其中很多内容可以用到Solr。

现在我们已经了解了Solr能够提供的一些优秀功能，接下来我们搭建并使用Solr。Solr将很多在Lucene中需要编程实现的环节转换成了可配置内容。

3.3.1 首次运行Solr

整个Apache Solr的源码和样例都包含本书附带源码包中。或者，也可以从Solr网站<http://lucene.apache.org/solr>上通过点击首页上的Download链接并按照下载说明进行下载。Solr需要Java JDK 1.6或更高版本。其随带Jetty Servlet容器，但是应该可以运行于大部分现代Servlet容器（如Apache Tomcat）。本书中，我们使用的是和本书源码绑定的版本，但是你也可以选择一个更新的版本，于是这里给出的操作说明可能会与你的说明有轻微不同。通过Solr网站可以得到正式的操作使用说明。在本书发布版本中，可以通过如下命令行给出的步骤来运行示例应用。

1. `cd apache-solr/example`
2. `java -jar start.jar`，该命令会启动Jetty，其中Solr会作为一个Web应用工作在端口8983。
3. 在浏览器中输入地址<http://localhost:8983/solr/>，你可以看到如图3-13所示的窗口。如果没有出现欢迎界面，那么参考Solr的网站来排除故障。
4. 在另一个命令行窗口下，和步骤1一样修改示例目录。
5. 在这个新命令行窗口下，输入 `cd exampledocs`。

6. 通过运行Java应用post.jar, 将目录下的示例文档发送给Solr: `java -jar post.jar *.xml`。
7. 切换你的浏览器和Solr欢迎界面, 点击Solr的管理界面 (如图3-14所示)。
8. 在查询框中, 输入并提交一个查询 (如Solr) 并浏览返回结果。

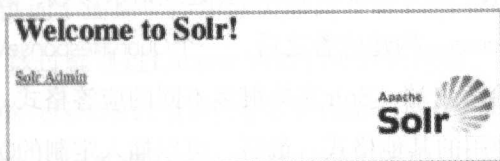


图3-13 Solr欢迎界面, 表明Solr应用正常启动

上述步骤就是Solr落地运行的全部过程。但是要为你的具体应用配置Solr的话, 必须构建Solr Schema (schema.xml), 并且有可能的话, 构建Solr配置文件 (solrconfig.xml)。出于本书的目的, 后续章节中只会强调这些文件的一部分, 但是完整的示例文件可以在本书附带源码的示例配置目录 (apache-solr/example/solr/conf) 下找到。此外, Solr网站 (<http://lucene.apache.org/solr>) 有很多介绍如何配置并利用Solr为特定需求服务的相关资源、入门讲座和文章。

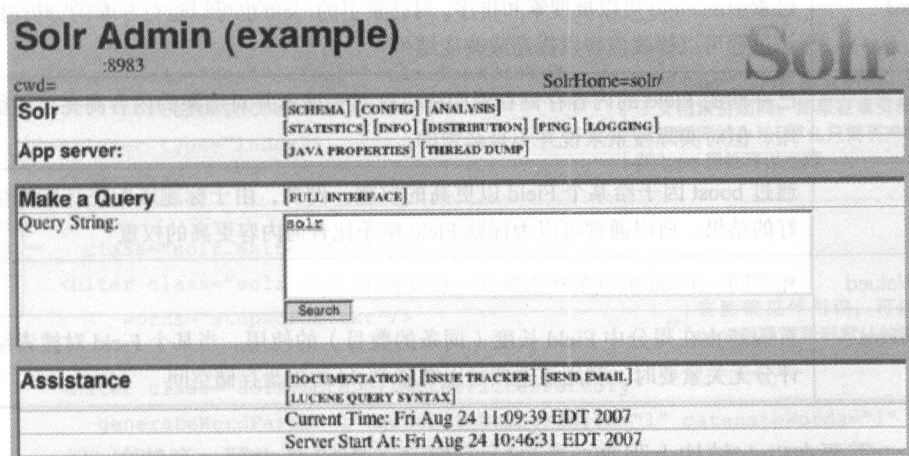


图3-14 Solr管理界面的截图

3.3.2 理解Solr中的概念

由于Solr是一个基于Web的搜索服务, 大部分操作都可以通过从客户端发送HTTP

GET和POST请求给Solr服务器端来完成。这种灵活性可以允许多种不同的应用与Solr一起使用，而不仅限于Java的应用。实际上，Solr提供了Java、Ruby、Python及PHP客户端代码，同时提供可便于任一应用来处理的标准XML应答。

当Solr收到客户端的一条请求时，它会分析URL并将该请求分发给合适的SolrRequestHandler。然后，SolrRequestHandler要处理请求参数，进行必要的计算，并组装出一个SolrQueryResponse。构建应答之后，一个QueryResponseWriter实现会对应答进行序列化处理并返回给客户端。Solr支持很多不同的应答格式，包括XML、JSON以及易于被Ruby和PHP使用的其他格式。最后，可以插入定制的QueryResponseWriters来根据需要提供可选的应答格式。

而对于内容处理，Solr采用了大量在Lucene索引和搜索中使用的相同术语和功能。在Solr和Lucene中，索引可以基于一个或多个Document来构建，每个Document包含一个或多个Field，而一个Field又由名字、内容以及要求Solr/Lucene使用的内容处理方式的元数据组成。这些元数据的描述如表3-3所示。

表 3-3 Solr 中的 Field 选项及其属性

名 字	描 述
Indexed	已索引的Field可以被搜索和排序。对已索引的Field也可以运行Solr的分析过程，该过程可以修改内容以提高或改变结果
Stored	已存储的Field的内容存储在索引中。这对于搜索并对结果的内容高亮显示很有用，但对实际搜索来说并不是必须的
boost	通过boost因子给某个Field以更高的权重。例如，由于标题匹配往往能产生更好的结果，所以通常可以为标题Field赋予比普通内容更高的权重
multiValued	允许同一Field可以多次添加到一篇文档中
omitNorms	有效禁止在得分中Field长度（词条的数目）的使用。当某个Field对搜索结果评分无关紧要时，利用该选项可以节省Field的磁盘存储空间

Solr需要在Field结构上强加一个schema定义（用XML书写，存储在schema.xml文件中），这样就可以通过FieldType的声明来允许Field的强类型，这一点要比Lucene更严格。这种方式下，可以声明某些Field是日期、整数或字符串类型以及Field所包含的属性。例如，dateFieldType声明如下。

```
<fieldType name="date" class="solr.DateField"      org.apache.solr.schema.DateField 的缩写
      sortMissingLast="true" omitNorms="true"/>
```

Solr同时也要求每个Document都有一个与其关键的独立Field值。

如果Field被索引，那么Solr可以应用一个分析过程来对该Field的内容进行转换。

这种方法下，Solr提供词干还原、停用词去除以及第3.2.1节介绍的将词条转换为索引词项的其他方法。该过程通过Lucene Analyzer类来控制。Analyzer由一个可选的CharFilter、一个必须的Tokenizer以及0个或多个TokenFilter构成。CharFilter可以用于在保留正确偏移信息（可用于高亮显示）的情况下去除内容信息（比如去除HTML标签）。注意在大部分情况下，都不需要CharFilter。Tokenizer用于生成Token，大部分情况下Token对应于要索引的单词。然后TokenFilter可以接受Tokenizer中给出的Token，然后可以选择在将它们传回给Lucene进行索引之前对它们进行修改或者去除处理。例如，Solr提供的WhitespaceTokenizer可以基于空格进行切词，而StopFilter可以从搜索结果中去除常见词。

作为一个更丰富的FieldType示例，Solr Schem样例（apache-solr/example/solr/conf/schema.xml）包含了如下文本类字段类型（注意：该文件可能会随时间变化，因此和这里可能并不完全一致）。

```
<fieldType name="text" class="solr.TextField"
      positionIncrementGap="100">
```

```
<analyzer type="index">
```

```
<tokenizer
```

```
class="solr.WhitespaceTokenizerFactory"/>
```

```
<filter class="solr.StopFilterFactory" ignoreCase="true"
```

```
words="stopwords.txt"/>
```

```
<filter class="solr.WordDelimiterFilterFactory"
```

```
generateWordParts="1" generateNumberParts="1" catenateWords="1"
```

```
catenateNumbers="1" catenateAll="0"
```

```
splitOnCaseChange="1"/>
```

```
<filter class="solr.LowerCaseFilterFactory"/>
```

```
<filter class="solr.EnglishPorterFilterFactory"
```

```
protected="protowords.txt"/>
```

正如 type="index" 所表明的，该分析器仅用于文档索引期间。如果在索引和查询处理期间使用相同的办法，那么只需要声明一个分析器，其 type 属性可以去掉

空格来生成词条。
每个 Tokenizer
TokenFilter 都包装
一个能够产生合适分
实例的工厂中

去除常见停用词，可以从 Solr
conf 目录下看到默认的停用词表

将包含混合大小写或数字等情况的单词进行
分割，比如，iPod 分成 iPod 以及 i 和 Pod

利用 Martin 博士的 Porter 工具对单词进行词
干还原处理。参考 <http://snowball.tartarus.org>

```
<filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer/>
</fieldType>
```

上例也表明，应用当中可以通过简单声明Tokenizer和TokenFilter的类型和次序来混合并提供与分析匹配的方法。

通过这两个FieldType的声明，可以以如下方式声明多个Field：

索引并存储
的日期Field
可以用于搜
索并按日期
对文档排序

```
<field name="date" type="date" indexed="true" stored="true"
      multiValued="true"/>
<field name="title" type="text" indexed="true"
      stored="true"/>
```

它们使用在 Solr 中声明的相
关 Analyzer 进行切分和分析

generator
Field 是 Solr
按输入完全一
样来索引和存
储的 StrField

```
<field name="generator" type="string" indexed="true"
      stored="true" multiValued="true"/>
<field name="pageCount" type="sint"
      indexed="true" stored="true"/>
```

将文档的 pageCount 存储为可
排序整数，这意味着该值对人
不可读但针对排序进行了优化

设计Solr的Schema

和任意文本分析系统的设计一样，必须要格外注意如何确保用户能够搜索。Solr的Schema语法提供了一个丰富的功能集合，从切词和词干还原的高级分析工具，到拼写检查和定制排序都具有可能性。所有这些功能都在Solr Schema（schema.xml文件）和Solr配置中指定。当开启一个新的Solr项目时，一个通用的规则是采用Solr中示例Schema和配置信息，并检查哪些要保留哪些要去除。由于示例Schema文档组织很好并仔细解释了每个概念，因此上述做法会很有效。

整个Solr Schema可以分成三个不同部分：

- Field类型声明
- Field声明
- 各种声明

Field类型声明部分告诉Solr该如何解释Field中包含的内容。然后，该声明中定义的类型可以用于后续的Field声明部分。但是由于类型声明并不意味着它一定会用。当前，Solr中包括很多类型，最常见的有IntField、FloatField、StrField、DateField及TextField等。此外，应用本身可以很容易实现自己的FieldType来扩展Solr

的类型功能。

Field声明部分是关键。在Schema的这部分当中，通过定义文档的名称、类型以及让Solr知道如何构建索引和处理搜索请求的元数据，应用可以清晰声明文档在Solr中的索引和存储方式。

最后，各种声明的部分包括各种混杂的声明，比如识别诸如Field的名称来作为文档或默认搜索字段的唯一键。另外的声明给出了Solr将某个Field的内容拷贝到另一个Field的方式。通过这种拷贝，Solr可以以多种方式高效分析同一内容，这样就可以在搜索时提供更多选择。例如，允许用户以大小写敏感的方式来搜索往往十分有用。通过构建一个<copyField>来将某个Field的内容拷贝到另一个保留大小写的Field，就可以容许大小写敏感的搜索。

在Schema设计中的一种诱惑往往是放入所有的东西，甚至包括那些无关紧要的东西，然后构建大型的查询来搜索所有的不同字段。尽管在Solr中可以实现这一点，更好的做法是想想哪些字段需要存储并索引，而哪些字段由于在别处存在而不需要这样处理。通常情况下，大部分简单查询通过构建一个包含所有其他可搜索Field的内容“all”Field就可以处理。采用这种策略，你可以在不需要生成跨Field查询的情况下实现内容的快速搜索。

在Solr中，分析过程很容易配置，往往不需要任何编程。只有在现有Lucene和Solr的CharFilter、Tokenizer和TokenFilter不够（人们已经提供了很多Lucene和Solr中的分析模块）的特殊情况下，才需要编写新的分析代码。

现在你已经了解Solr如何对内容进行结构化的基本知识，我们可以进一步深入了解如何将内容添加到Solr以便可以对它们进行搜索。下一节将介绍如何对Document进行明确表示并将它们输送给Solr进行索引，之后我们会考察如何对这些内容进行搜索。

3.4 利用Apache Solr对内容构建索引

Solr有多种方法来对内容进行索引，包括从XML或JSON消息、CSV文件或常见办公MIME类型文件中读取信息，或通过SQL命令从数据库或从RSS源中读取信息。下面我们主要介绍使用XML消息和常见办公文件格式的索引方法，而其他方法可以参考Solr文档。特别地，如果需要了解更多有关CSV文件索引的信息，请参考<http://>

wiki.apache.org/solr/UpdateCSV。最后，要学习更多有关数据库和RSS源索引的知识，请参考<http://wiki.apache.org/solr/DataImportHandler>的Data Import Handler。

在讨论XML索引之前，需要注意在Solr中有如下四种类型的索引行为。

- 添加/更新：允许添加或更新一篇文档到Solr。这种添加和更新只有提交之后才能在搜索中有效。
- 提交：通知Solr上次提交之后做出的修改可以为搜索所用。
- 删除：可以基于ID或查询删除文档。
- 优化：重构Lucene内部结构来提高搜索的性能。而如果可以实现优化的话，最好在索引结束之后再进行。在大部分情况下，你不用考虑优化的事情。

3.4.1 使用XML构建索引

Solr中的一种索引方法涉及从预处理的内容构建XML消息并将它作为HTTP POST消息进行发送。这条XML消息如下所示：

```
<add>
  <doc>
    <field name="id">solr</field>
    <field name="name" boost="1.2">
      Solr, the Enterprise Search Server
    </field>
    <field name="mimeType">text/xml</field>
    <field name="creator">Apache Software Foundation</field>
    <field name="creator">Yonik Seeley</field>
    <field name="description">An enterprise-ready, Lucene-based searchserver.
      Features include search, faceting, hit highlighting, replication and
      much, much more</field>
  </doc>
</add>
```

在上述示例XML中，可以看到一个简单的结构，该结构首先声明了<add>命令，然后包含了一条或多条<doc>记录。每篇文档指定与之关联的Field及相应的可选boost值。之后，这条消息可以作为任意Web浏览器或HTTP客户端的POST命令发布给Solr。要了解更多Solr中利用XML命令的信息，请参考<http://wiki.apache.org/solr/UpdateXmlMessages>的Solr wiki内容。

幸运的是, Solr有一个易用的称为SolrJ的客户端库,它能够处理所有构建Solr XML消息所涉及的工作。程序清单3-1展示了如何使用SolrJ将文档加入到Solr的过程:

清单 3-1 SolrJ客户端库的使用示例

```

一条基于 HTTP Solr 服务器连接 → SolrServer solr = new CommonsHttpSolrServer (
                                new URL ("http://localhost:" + port + "/solr"));

SolrInputDocument doc = new SolrInputDocument ();

doc.addField ("id", "http://tortoisehare5k.tamingtext.com"); ← Solr 的这个实例所用的
                                                                    Schema 需要一个唯一的
                                                                    字段名称 ID

doc.addField ("mimeType", "text/plain");

doc.addField ("title",
                "Tortoise beats Hare! Hare wants rematch.", 5); ← 将标题字段加入文档并赋予其
                                                                    一个 5 倍于其他字段的权重

必须在 Solr 中  Date now = new Date ();
或某种特定格式 → doc.addField ("date",
                                DateUtil.getThreadLocalDateFormat ().format (now));

doc.addField ("description", description);

doc.addField ("categories_t", "Fairy Tale, Sports"); ← 一个允许在 Solr 中加入未知字
                                                                    段的动态字段。_t 通知 Solr 应
                                                                    该将它看作文本字段来处理

所有文档加入  solr.add (doc); ← 将新创建的文档发送给 Solr。Solr 注意创建一
用希望能够搜索 个正确格式的 XML 消息并使用 Apache Jakarta
们之后, 给  Commons HttpClient 将它发送给 Solr
发送一个提  solr.commit ();
息

```

为索引内容, 必须要为每个SolrInputDocument发送Solr add命令。注意多个SolrInputDocument可以包含在单个add命令中, 这只需要使用SolrServer的add方法即可完成, 该方法的输入是Collection。为了提高性能, 鼓励这样做。你可能考虑HTTP的开销可能会对索引性能造成影响, 但现实是, 处理连接所需要的工作量在大部分情况下相比索引开销而言都很小。

现在你已经了解利用XML进行Solr索引的基本知识。下面考察如何对常见格式文件构建索引。

3.4.2 利用Solr和Apache Tika对内容进行抽取和索引

为抽取内容并在Solr中建立索引，需要使用本章和第1章介绍的多个概念，它们不仅仅是在用户界面给出一个搜索框。首先，必须要设计一个能够反映搜索内容信息的Solr Schema。由于你使用Tika来实现抽取功能，应用本身必须要将Tika生成的元数据和Tika生成的内容映射到自己的Schema字段中。

为浏览完整的Solr Schema，你可以在任何最喜欢的编辑器中打开solr/conf/schema.xml文件。除了确保将Tika中正确的类型映射为Solr中合适的FieldType外，不需要在Schema的设计因素中加入更多信息。例如，页数是一个整数，因此在Solr Schema中作为整数。对于该字段及其他字段，我们反复浏览文档样本以及基于它们抽取的结果。为实现这一点，我们利用Solr、Tika集成之后可以抽取内容而不索引这个事实，利用一个叫curl的工具（可以从大部分装有*NIX系统的机器上获得，对于Windows系统从<http://curl.haxx.se/download.html>下载），你可以发送文件与其他HTTP请求给Solr。如果Solr没有运行，利用第1章的命令来启动。一旦Solr启动运行，就可以用它来索引某些内容。在这个例子中，我们将下列命令发送给Solr以抽取示例文件中的内容：

```
curl "http://localhost:8983/solr/update/extract?&extractOnly=true" \
-F "myfile=@src/test/resources/sample-word.doc"
```

上述命令的输入是一篇位于本书源码src/test/resources目录下的样例Word文档，然而你也可以尝试任意一篇Word文档或PDF文件。对curl而言，要正确定位样本文档，必须要从本书源码的根目录下运行curl命令。最后的输出结果会包含抽取的内容以及元数据，看上去如下所示（为了简洁起见，去掉了部分内容）：

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">8</int>
  </lst>
  <str name="sample-word.doc">&lt;?xml version="1.0"
    encoding="UTF-8"?&gt;
    &lt;html xmlns="http://www.w3.org/1999/xhtml"&gt;
    &lt;head&gt;
```

```
<title>This is a sample word document</title>
</head>
<body>
<p>This is a sample word document.&#xd;
</p>
</body>
</html>
</str>
<lst name="sample-word.doc_metadata">
  <arr name="Revision-Number">
    <str>1</str>
  </arr>
  <arr name="stream_source_info">
    <str>myfile</str>
  </arr>
  <arr name="Last-Author">
    <str>Grant Ingersoll</str>
  </arr>
  <arr name="Page-Count">
    <str>1</str>
  </arr>
  <arr name="Application-Name">
    <str>Microsoft Word 11.3.5</str>
  </arr>
  <arr name="Author">
    <str>Grant Ingersoll</str>
  </arr>
  <arr name="Edit-Time">
    <str>600000000</str>
  </arr>
  <arr name="Creation-Date">
    <str>Mon Jul 02 21:50:00 EDT 2007</str>
  </arr>
  <arr name="title">
    <str>This is a sample word document</str>
  </arr>
  <arr name="Content-Type">
    <str>application/msword</str>
  </arr>
  <arr name="Last-Save-Date">
    <str>Mon Jul 02 21:51:00 EDT 2007</str>
  </arr>
```

```
</lst>
</response>
```

从上述输出结果中，我们可以看到Tika返回的信息类型，然后基于该类型来规划你自己的Schema。

Schema定义之后，通过将文档发送给Solr的ExtractingRequestHandler来处理索引过程，ExtractingRequestHandler提供了必要的基础构架来利用Tika进行内容抽取。为了使用ExtractingRequestHandler，必须在Solr配置文件中对它进行配置。在本例中，solrxonfig.xml包含下列信息：

```
<requestHandler name="/update/extract"
  class="org.apache.solr.handler.extraction.ExtractingRequestHandler">
  <lst name="defaults">
    <str name="fmap.Last-Modified">last_modified</str>
    <str name="fmap.Page-Count">pageCount</str>
    <str name="fmap.Author">creator</str>
    <str name="fmap.Creation-Date">created</str>
    <str name="fmap.Last-Save-Date">last_modified</str>
    <str name="fmap.Word-Count">last_modified</str>
    <str name="fmap.Application-Name">generator</str>
    <str name="fmap.Content-Type">mimeType</str>
    <!-- Map everything else to ignored -->
    <bool name="uprefix">ignored_</bool>
  </lst>
</requestHandler>
```

由于上述信息已经打包到tamingText-src Solr的设置当中，因此你已经完成构建索引所需的设置。此时此刻，所有需要你做的就是发送一些文档给Solr。为了达到自己的目标，我们再次需要依赖curl，但是对于真实应用来说，采集器或者一段从存储数据集（CMS、DB等）中读取文档，并通过SolrJ等Solr客户端发送给Solr的代码会更合适。

为演示通过curl处理的结果，你可以通过去除extract.only命令中的extract.only参数并增加一些其他参数：

```
curl "http://localhost:8983/solr/update/extract?
  literal.id=sample-word.doc&defaultField=fullText&commit=true" \
  -F myfile=@src/test/resources/sample-word.doc
```

除了指定要上载的文件（-F 参数）之外，上述命令中有以下两个非常重要的部分。

- 发送文件给 /update/extract URL而不是 just/update。这表明你希望使用前面配置的ExtractingRequestHandler。
- 本例当中你输入的参数包括
 - literal.id=sample-word.doc：这告诉Solr将literal的值sample-word.doc以Field方式加入到Document中，命名为ID。换句话说，这就是你的唯一ID。
 - defaultField=fullText：有可能的话，Solr会自动将抽取的内容名称和Solr Field名称相匹配。如果没有任何名称匹配，那么该值指定的是索引内容所需要的默认字段。在本例中，所有无映射及无匹配的内容会进入fullText字段。
 - commit=true：这要求Solr立即提交新文档到索引以便能够搜索这篇文档。

上面的命令覆盖了ExtractingRequestHandler的基本知识，要了解更高级的功能，请参考<http://wiki.apache.org/solr/ExtractingRequestHandler>。

对你的文档集运行索引命令得到的结果应该比较类似，当然可以按照你目录下的文档进行调整。给定这个简单的索引之后，你可以通过Solr的内置管理工具对索引进行查询看看能够得到什么结果。

小技巧 当使用Lucene和Solr的索引时，Luke是理解索引内容的最好工具。该工具由Andrzej Bialecki编写，有助于了解词项索引的过程以及索引了哪些文档，另外还可以了解其他的一些元数据信息，比如某个字段中频率最高的50个词项。Luke可以从<http://code.google.com/p/luke/>免费下载。

现在索引中已有一些内容，接下来我们来了解Solr中最关键的部分，即搜索如何工作。在下一节中，将使用SolrJ来给Solr发送搜索请求并浏览返回结果。

3.5 利用Apache Solr来搜索内容

和索引过程很像，搜索过程也通过向Solr发送包含用户需求的HTTP请求来实现。Solr包含一个富查询语言，能够允许用户使用词项、短语、通配符和其他一

些选项来完成查询。这些选项随着每次新版本的发布而增长。读者不必对此心存忧虑，Solr网站（<http://lucene.apache.org/solr>）在文档方面做了很好的工作，所有的新功能都进行了全面描述，因此可以查阅到最新最强大的查询功能。

为理解Solr中的搜索过程，我们后退一步看看Solr是如何处理请求的。如果你还记得3.3.2节内容，Solr检查输入的消息并将请求路由到SolrRequestHandler接口的一个实例。幸运的是，Solr自带了很多有用的SolrRequestHandler，这样你就不必自己来实现。表3-4列出了一些最常见的SolrRequestHandler及其功能。

表 3-4 常见 SolrRequestHandler

名 称	描 述	查 询 示 例
StandardRequestHandler	正如从名称可以猜到的那样，StandardRequestHandler 是 默 认 的 SolrRequestHandler。其提供了指定查询词项、搜索字段、返回结果数目、多面、高亮及相关反馈等机制	&q=description%3Awin+OR+description%3Aall &rows=10 对 description 字段进行查询，看其包含 all 还是 win，最多返回 10 行结果
MoreLikeThisHandler	返回与给定文档相似的文档	&q=lazy&rows=10&qt=%2Fm lt&qf=title^3+des cription^10
LukeRequestHandler	LukeRequestHandler 的 命 名 参 考 了 Lucene/Solr 索引发现快捷工具 Luke。Luke 是一个简单但是强大的 GUI 工具，它可以提供对现有索引结构和内容的深入了解。LukeRequestHandler 模拟了 Luke 的大部分功能，它通过查询应答的形式提供有关索引的元数据，这些查询应答可以用于应用来展示索引的信息	&show=schema 返回当前索引的 Schema 信息（字段名、存储和索引状态等）

尽管RequestHandler一般可以处理任意请求，在实际处理搜索请求时常常使用一个继承类SearchHandler。SearchHandler由多个SearchComponent构成。而SearchComponent加上查询分析器在搜索时负责完成大部分重要工作。Solr有多种SearchComponent，同时也提供了一些不同的查询分析器。这些模块都是可插入的，你也可以加入自己的搜索组件或者查询分析器。为更好理解这个问题以及其他Solr输入参数，下面我们先近距离考察一下这些功能。

3.5.1 Solr查询输入参数

Solr提供了丰富的语法来表示输入参数和输出处理方式。表3-5给出了7种不同的输入类别，并给出了详细描述及常见或有用的参数。由于Solr常常升级，因此可以通过查阅Solr网站来获得权威的参数列表。

表 3-5 常见 Solr 输入参数

键	描 述	默认值	支 持 类	例 子
q	实际查询。基于所用 SolrRequestHandler 的不同语法有所变化。对于 StandardRequestHandler 来说，所支持的语法在 http://wiki.apache.org/solr/SolrQuerySyntax 有描述	N/A	StandardRequestHandler, DisMaxRequestHandler, MoreLikeThisHandler, SpellCheckerRequestHandler	q=title:rabbit AND description:"Bugs Bunny" q=jobs:java OR programmer
sort	指定结果的排序 Field	score	大部分 SolrRequestHandler	q=ipod&sort=price desc q=ipod&sort=price desc, date asc
start	返回结果开始的偏移位置	0	大部分 SolrRequestHandler	q=ipod&start=11 q=ipod&start=1001
rows	返回的结果数目	10	大部分 SolrRequestHandler	q=ipod&rows=25
fq	指定一个 FilterQuery 来限制返回结果。FilterQuery 十分有用，比如，当将结果限制到某个时间范围内或者所有标题都包含一个 A。FilterQuery 只在限制集合上反复查询时才有用	N/A	大部分 SolrRequestHandler	q=title:ipod&fq=manufacturer:apple
facet	请求给定查询的多面信息	N/A	大部分 SolrRequestHandler	q=ipod&facet=true
facet.field	指定多面的 Field。该 Field 用于构建多面集合	N/A	大部分 SolrRequestHandler	q=ipod&facet=true&facet.field=price&facet.field=manufacturer

自然而然地，如何在你的应用中进行取舍表3-5中的这么多选项以及在线的更多选项是一个难题，关键是得了解你的用户及其搜索方式。通常而言，像高亮显示和 More Like This 这种尽管看上去很好的功能会因为额外处理（特别是上述功能进行组合时）降低搜索的速度。另一方面，当用户希望快速定位匹配内容来集中关注答案时高亮显示就很有用。接下来看看如何通过编程方式访问 Solr，因为这是将 Solr 集成到你的应用程序的主要方式。

注意 上述有关输入参数的所有讨论无疑会让你想知道这些工作带来的回报。也就是说，结果看起来怎样？你怎样对它们进行处理？为此，Solr 提供了一个可插入的结果处理 handler 类，该类继承了 `QueryResponseWriter`。与存在的 `SolrRequestHandler` 多个实现类似，`QueryResponseWriter` 也有多种实现，总有一种能够满足你的输出需求。最常见（也是默认）的应答类是 `XMLResponseWriter`，其负责序列化搜索、多面并对结果高亮显示（以及其他功能）并以客户端可以处理的 XML 格式返回结果。其他的实现还包括 `JSONResponseWriter`、`PHPResponseWriter`、`PHPSerializedResponseWriter`、`PythonResponseWriter`、`RubyResponseWriter` 和 `XSLTResponseWriter`。我们希望这些实现的名称就能完美表明各自的功能，你也可以通过 Solr 的网站来获取更多的细节。此外，如果需要与遗留系统交互或输出你自己的二进制格式，那么实现一个 `QueryResponseWriter` 就是相对比较直接的方法，在 Solr 的源码中给出了很多例子。

编程访问 Solr

迄今为止，你已经看到 Solr 的多种不同输入，但是实际执行搜索的代码到底如何？

Solr 也提供了一些更高级的尽管看上去有些普通的查询功能。例如，`DismaxQParser` 查询分析器提供了一种比 `LuceneQParser` 查询分析器更简单的语法，并能给予句出现在不同字段的文档赋予优先级。程序清单 3-2 的样例代码展示了如何调用 `DismaxQParser` 来进行查询分析的 `RequestHandler`。

清单3-2 Solr查询代码样例

Max 分析器
给定 qf 参数
多个字段间进
搜索并提升相
同项的权重

```
queryParams.setQuery ("lazy") ;
queryParams.setParam ("defType", "dismax") ;
queryParams.set ("qf", "title^3 description^10") ;
System.out.println ("Query: " + queryParams) ;
response = solr.query (queryParams) ;
assertTrue ("response is null and it shouldn't be", response != null) ;
documentList = response.getResults () ;
assertTrue ("documentList Size: " + documentList.size () +
    " is not: " + 2, documentList.size () == 2) ;
```

通知 Solr 使用 DisMax 查询分析器 (在 solrconfig.xml 中的名称为 dismax)

另一种常见的搜索技术是为用户提供一种快速简便的方法来访问与当前结果文档相似的文档。该过程常常称为Find Similar或More Like This。Solr内置了More Like This功能，它们只需要配置即可使用。例如，在solrconfig.xml中，可以以如下方式指定 /mlt请求处理类：

```
<requestHandler name="/mlt" class="solr_MoreLikeThisHandler">
  <lst name="defaults">
    <str name="mlt.fl">title, name, description, fullText</str>
  </lst>
</requestHandler>
```

在上述简单的配置中，你指定的MoreLikeThisHandler要使用title、name、description和fullText字段作为新查询的产生源。当用户请求一个More Like This查询时，Solr会检查输入文档，在指定字段查找词项，计算其中最重要的部分而后生成新的查询，之后新查询会提交给索引并返回结果，为查询新请求处理类，可以使用如程序清单3-3所示代码。

清单3-3 More Like This示例代码

要计算相似
结果文档

```
queryParams = new SolrQuery () ;
queryParams.setQueryType ("/mlt") ;
queryParams.setQuery ("description:number") ;
queryParams.set ("mlt.match.offset", "0") ;
queryParams.setRows (1) ;
queryParams.set ("mlt.fl", "description, title") ;
response = solr.query (queryParams) ;
assertTrue ("response is null and it shouldn't be", response != null) ;
SolrDocumentList results =
```

构建搜索来寻找相似文档

指定字段来生成查询

```
(SolrDocumentList) response.getResponse ().get ("match") ;
assertTrue ("results Size: " + results.size () + " is not: " + 1,
results.size () == 1) ;
```

Solr/Lucene Statement	search
Start Row	0
Maximum Rows Returned	10
Fields to Return	*.score
Query Type	dismax
Output Type	standard
Debug: enable	<input type="checkbox"/> <small>Note: you may need to "view source" in your browser to see explain() correctly indented.</small>
Debug: explain others	<input type="checkbox"/> <small>Apply original query scoring to matches of this query to see how they compare.</small>
Enable Highlighting	<input type="checkbox"/>
Fields to Highlight	
<input type="button" value="Search"/>	

图3-15 Solr查询界面

3.5.2 抽取内容的多面展示

在3.4.2节，我们利用Solr和Tika对某些样本MS Word文件构建了索引。这一部分，我们会加入更多自己的内容，因此你的结果可能根据内容不同而不同。但是你既然已经对如何利用SolrRequestHandler进行搜索有更深入的理解，就可以使用Solr的简单管理查询界面来运行一系列搜索。在先前的例子中，我们让Solr运行在本机的8983端口。在Web浏览器中访问http://localhost:8983/solr/admin/form.jsp会得到与图3-15类似的结果。

在本例中，我们使用dismax SolrRequestHandler，它定义在solrconfig.xml配置文件中，如下所示：

```
<requestHandler name="dismax" class="solr_DisMaxRequestHandler" >
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <float name="tie">0.01</float>
    <str name="qf">
      name title^5.0 description keyword fullText all^0.1
    </str>
    <str name="fl">
      name, title, description, keyword, fullText
    </str>
  </lst>
</requestHandler>
```

```

</str>
<!-- Facets -->
<str name="facet">on</str>
<str name="facet.mincount">1</str>
<str name="facet.field">mimeType</str>
<str name="f.categories.facet.sort">true</str>
<str name="f.categories.facet.limit">20</str>
<str name="facet.field">creator</str>
<str name="q.alt">*:*</str>
<!-- example highlighter config, enable per-query with hl=true -->
<str name="hl.fl">name, title, fullText</str>
<!-- for this field, we want no fragmenting, just highlighting -->
<str name="f.name.hl.fragsize">0</str>
<!-- instructs Solr to return the field itself if no query terms are
found -->
<str name="f.name.hl.alternateField">name</str>
<str name="f.text.hl.fragmenter">regex</str><!-- defined below -->
</lst>
</requestHandler>

```

为使查询过程更简单，我们已经在dismax查询处理类中放入很多默认值来指定需要搜索并返回哪些字段，同时也指定哪些字段要多面和高亮展示。提交示例查询给Solr之后，我们会得到一个包含结果的XML文档以及多面信息。

清单3-4 Solr搜索结果示例

```

<response>

<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">4</int>
  <lst name="params">
    <str name="explainOther"/>
    <str name="fl">*, score</str>

    <str name="indent">on</str>
    <str name="start">0</str>
    <str name="q">search</str>
    <str name="hl.fl"/>
    <str name="wt">standard</str>
    <str name="qt">dismax</str>

```

应答头部返回有关输入参数和搜索的元数据

```

<str name="version">2.2</str>
<str name="rows">10</str>
</lst>
</lst>
<result name="response" numFound="2" start="0"
      maxScore="0.12060823">
  <doc>
    <float name="score">0.12060823</float>
    <arr name="creator">...</arr>
    <arr name="creatorText">...</arr>
    <str name="description">An enterprise-ready, Lucene-based
      search server. Features include search,
      faceting, hit highlighting, replication and much,
      much more</str>
    <str name="id">solr</str>
    <str name="mimeType">text/xml</str>
    <str name="name">Solr, the Enterprise Search Server</str>
  </doc>

  <doc>
    <float name="score">0.034772884</float>
    <arr name="creator">...</arr>
    <arr name="creatorText">...</arr>
    <str name="description">A Java-based search engine library
      focused on high-performance, quality results.</str>
    <str name="id">lucene</str>
    <str name="mimeType">text/xml</str>
    <str name="name">Lucene</str>
  </doc>
</result>

<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="mimeType">
      <int name="text/xml">2</int>
    </lst>
    <lst name="creator">
      <int name="Apache Software Foundation">2</int>
      <int name="Doug Cutting">1</int>
      <int name="Yonik Seeley">1</int>
    </lst>
  </lst>

```

<result> 给出了和查询匹配的文档信息、配置信息和输入参数

```
</lst>
<lst name="facet_dates"/>
</lst>
</response>
```

Facet 字段列表提供搜索结果中发现的多面的细节信息。在本例中, mimeType facet 表明 9 个结果中的 4 个是 image/tiff, 另外 4 个是 text/plain, 还有一个是 image/png

清单3-4给出了针对示例查询的缩减结果集合(为了显示,有些字段信息故意丢掉),同时包括多面信息。尽管我们依赖于dismax配置中指定的默认值,你可以通过URL来传入参数以覆盖默认参数,从而返回更多结果、高亮信息或者查询不同字段。

到此为止,我们大致覆盖了开始使用Solr的核心知识。对Solr网站进行探索会提供更多所有主题的细节,并提供更多先进的主题,如缓存、复制和管理等等。接下来我们回退一步来考察一般的性能问题,然后关注Solr中的一些特定性能问题,最后了解这些因素如何影响你的搜索实现。

3.6 理解搜索性能因素

从高层次上来说,搜索性能指的是搜索系统返回结果的质量指标。它可以进一步分成两类:数量和质量。数量指的是返回结果的速度(在给定时间内返回结果的数目),而质量指的是搜索结果的相关程度。通常情况下(并不总是)上述两个方面是互相对立的,需要实践人员不断评估以便在更快的速度和更好的结果之间折中。本节会考察现代搜索引擎用于同时提高数量和质量的技术和技术。之后考察针对Solr的一些提高性能的微调办法。在这些主题之前,我们快速浏览一遍如何判定速度和相关性,这是因为如果不了解这些,就无法知道搜索是否已经取得成功。

3.6.1 数量判定

作为搜索引擎的用户,输入查询之后如果存在返回结果的话,那么没有比返回结果只有微弱相关性更让人沮丧的了。接下来往往是快速浏览返回的前10个结果然后增加或删除关键词来试错。通常地,如果刚好能够得到正确的关键词组合,那么就可以找到想要的结果。否则的话,感觉应该放弃尝试。

另一方面,搜索引擎创建者们常常在结果质量、易用性和速度之间努力折中。由于本质上来说,查询是用户信息需求的不完整表达形式,搜索引擎往往通过复杂的算法来试图填补用户不完整查询和其真实需求之间的鸿沟。

在用户和搜索引擎创建者之间有一个模糊的称为相关度的概念。相关度指某个结果和用户查询的吻合程度。之所以模糊是因为任意两个用户对于是否吻合的看法都不会在所有情况下完全一致。尽管相关度判定是一件十分主观的任务，很多人已经开始尽力通过系统性的方法来确定相关度。有些甚至已经开始组织会议，通过标准文档集、查询和评估工具来完成相关性判定任务。会议参加人员在文档集上运行查询并将结果提交给会议组织方，然后会议组织方收集大家的结果进行评估并按照结果的相关度高低排序。这些会议最早来自一年一度的文本检索会议（text retrieval conference, TREC），该会议由美国国家标准技术局（national institute of standards and technology, NIST）组织举办。

在确定到底哪家引擎的结果最好时，上述会议中很多都依赖于两个评价指标。第一个指标称为正确率，指的是引擎返回结果当中相关文档所占的比例。在此基础上进行改进的一个常用方法是只考察部分返回结果的上下文。例如，前10个结果的正确率（通常记为 $P@10$ ）度量的是前10个返回结果中相关文档所占的比例。由于大部分用户只浏览前10个或者第一页的结果，只考察前10个结果的正确率往往十分有用。第二个指标称为召回率，指的是文档集所有相关文档中被检索出的比例。注意，对每条查询返回文档集中所有文档便可以得到最优召回率，当然这种做法显然十分愚蠢。很多情况下，必须要在正确率和召回率之间折中。例如，要求查询中的更多词项（如果不可能全部词项的话）出现在文档当中往往可以提高文档的正确率。但是在小规模文档集上，这可能意味着没有任何返回结果。类似地，通过将查询中每个词项的任意一个或者全部同义词加入到查询中往往可以提高召回率。不幸的是，由于很多词都有歧义，包含词的其他含义的文档可能会出现在最终结果中，因此会降低正确率。

因此，如何评估系统的质量？首先，要认识到每项技术都有优缺点。多半情况下，需要使用一种以上技术。表3-6列出了一些评估技术，毋庸置疑，不在此表中的一些其他技术对于特定系统来说可能也被证实十分有用。

表3-6 常见评估技术

技 术	描 述	开 销	权 衡
Ad hoc	开发者、质量保证人员和其他方非正式对系统进行评估并提供反馈来指出哪一环节可行或不行	一开始低,但是长期来看可能会很高	除非保留了日志信息,否则很难正式重现。很难知道所做的修改如何影响系统的其他部分。或许重点关注效果方面过于狭窄。有对某些具体查询集调优的风险。至少,所有测试者应该尝试相同的测试文档集
焦点小组 (Focus group)	分组用户被邀请使用系统。所用的查询、选择的文档等信息被记录。用户可以被显式要求识别相关和不相关文档。然后,统计信息可以用于对搜索质量进行决策	取决于参与者的数量和搭建评估系统的开销	可能有用,这取决于参与者的数量。也可以提供有关可用性的相关反馈。日志被保存用于构建可重复测试
TREC 及其他评价会议	TREC 提供了一系列任务来评估信息检索系统,其中有些任务关注 Web、博客或法律类文档	获取数据需要付费,正式参加(提交结果)TREC 可能费时费力。问题和判定结果可以免费获取,利用这些数据可以根据需要离线运行系统并评估	优点是能和其他系统比较并获得有关质量的总体感觉,但是数据可能不能代表你的文档集,因此结果对于你的系统来说可能不是那么有用

续表

技 术	描 述	开 销	权 衡
日志分析	从生产系统中获取日志，抽取排名最高的 50 个查询以及 10 到 20 个随机样本。分析并判定每个查询的前 5 或 10 条结果中哪些强相关、弱相关、不太相关以及完全不相关。对于表现较差的查询的结果进行关联和分析。随着时间的推移，目标是最大化相关文档数目、最小化不相关文档数目并去掉那些完全不相关的文档。此外，要对那些返回零相关结果的搜索进行分析以提高结果。进一步的分析可能涉及哪些结果被点击（点击分析）、用户在每个网页上花费多长时间。这其中的假设在于，用户在给定结果上花费的时间越多，那么该结果与他们的需求也越相关	取决于日志的规模和所用查询的数目	特别适合于在你的数据上测试你的用户。需要对记录什么何时进行搜索进行有效规划。最好在 β 测试中使用并作为生产系统的一项持续性工作进行。风险在于早期可能出现低质量结果
A/B 测试	和查询日志分析和焦点小组测试类似，使用一个活动系统，但是不同用户可能会收到不同结果。例如，有 80% 的用户会收到某种方法的结果，而剩下 20% 的用户可能收到另一种方法的结果。然后，分析日志并进行比较，以确定使用相同查询的两组用户是倾向于使用哪种方法。这种测试方法可以用于系统的任意部分，包括索引如何构建到结果如何展示。要确保完整记录 A 组用户和 B 组用户的差异	需要在生产中部署和支持两套系统。开销也取决于查询的数量和日志的规模	非常适合于实际用户的测试。最好在某段非高峰受限时间段完成。换句话说，如果测试的是一个销售网站，那么千万不要在圣诞节的三周前进行测试

Ad hoc、焦点小组、TREC以及使用范围更小一点的日志分析和A/B测试都存在一种风险，就是可能会产生一个对于评估来说最优但是在实际当中并非如此的系统。例如，利用TREC文档、查询和评估结果（也叫相关性判断）可能意味着你的系统在TREC风格的查询（具有一定特性）上取得了很好的结果，但是在用户提交的查询上可能效果并不好。

从实际的角度来说，大部分面向大量用户的应用至少进行Ad hoc测试和查询日

志分析。如果资金足够的话，焦点小组和TREC形式的评估可以给你评估系统以更多的数据点。日志分析和A/B测试会产生最实际的可用结果，是本书作者们首选的方法。

自然而然，相关性调优永远不会只是一次性的任务。同样，如果某个查询不是最频繁的查询之一的話，你也不应该纠缠于该查询结果的质量。此外，你应该很少担心为什么有条结果排名第4而另一条排名第5。或许唯一需要担心的时候是编辑化决策（某人为某个位置付费）需要某篇文档排在一个特定的位置。在一天结束时，如果某结果是某个查询的最高命中文档，那么该命中次数就是当前查询的硬编码结果。试图通过调整系统的不同配置参数以让某条结果在看上去正常搜索的结果中排名第一是一件令人头痛的事情，同时也更有可能破坏其他查询的搜索过程。知道何时用锤子何时用螺丝刀的话，就让你的客户保持满意这场战争胜出了一半。

3.6.2 判断数量

有很多指标可以用于判断搜索系统在数量方面的性能。其中一些有用的指标列举如下。

- 查询吞吐率：给定时间单位内系统能够处理的查询数目。通常以每秒内处理的查询数（QPS）来度量，该值越高越好。
- 流行度vs. 时间：用图表的方式给出每条查询持续的平均时间及频率。该指标在展示需要在系统何处花费时间来提高系统性能时尤其有用。例如，如果最流行的查询的频率同时最低，那么系统就有问题。但是如果只有那些罕见查询频率较低，那么可能就不值得花费精力进行考察。
- 平均查询时间：查询的平均处理时间。相同的统计信息可以展示查询随时间的分布。该值越小越好。
- 缓存统计量：很多系统会缓存查询结果和文档，了解缓存命中的频率十分有用。如果常规条件下命中次数很少，那么关掉缓存之后可能速度更快。
- 索引规模：度量索引算法在压缩索引方面的效果。有些索引可能是原始索引的20%。该值越小越好，但是由于磁盘很便宜因此不必对此过于担心。
- 文档吞吐率：每个时间单位内索引的文档数目。通常用每秒处理的文档数

（DPS）来衡量。该值越高越好。

- 索引中的文档数目：当索引规模变得太大时，可能需要采用分布式存储。

- 独立词项数目：一种高级指标，可以提供对索引规模的基本了解。

此外，对诸如CPU、RAM、磁盘空间和I/O消耗的常规检测对于判断系统的性能也起重要作用。这些指标的关键是必须要随时间推移进行监测。很多系统（包括Solr）提供了一个管理工具，能够让管理员对于资源使用情况得心应手。

了解系统运行性能的基本知识后，下一节将考察多种能够实际提高性能的技巧和技术。这一节会在一个较宽范围内考察多种技术，从考虑硬件到在多种搜索和索引选项之间折中。

3.7 提高搜索性能

从早期搜索引擎开始，研究人员和实际开发人员就为多种目的来调节系统。有些希望更高的相关度，有些则希望更高的压缩率，还有一些希望有更高的查询吞吐率。现在，我们希望达到上述所有目标，但是如何才能做到这一点？接下来的内容希望能够为系统搜索性能提供一系列选项供思考和尝试。

在开始之前，先给出一个警告：对搜索引擎进行调优需要花费大量时间，但是结果的提高可能无法测量。最好在对系统进行调优之前，要对监督质量和数量两方面所需要的改进都进行检查。另外，这里提到的建议不可能在所有情况下都有用。根据搜索引擎的不同，这里提到的某些或者全部技巧甚至是不适用的，或者需要在实现之前对搜索引擎有很深的理解。最后，在花太多时间调优之前，要确保问题确实出在搜索引擎而不是应用中。

带着上述指标和给出的警告，下面考察如何才能提高搜索性能。有些性能问题属于索引时的问题，而有些则属于搜索时的问题。有些问题仅仅与结果的数量或质量有关，而有些则会影响两个方面。下面首先看看有关硬件的问题，然后再讨论提高分析能力和提交更好查询的软件解决方案。

3.7.1 硬件改进

对于所有搜索引擎来说，最容易也是花费最大的调优办法就是升级硬件。搜索引擎常常希望大量RAM，并希望CPU都为它们服务。此外，大型系统内

存完全不够的情况下，I/O系统的改进也会带来收获。查询端的一个特别兴趣集中于固态硬盘（SSD），它可以大幅度减少寻道时间，但是也可能会降低写数据的速度。

单机的提高只能到此为止。某种情况下，这取决于数据和机器的规模及用户的数目，有时整个工作负载必须要分布到两台或多台机器上去。此时通常可以采用如下两种方法的一种进行处理。

1、复制：能够在单台及其内存中存放的单个索引复制到一台或多台负载均衡的机器上去。这对于在线商店来说比较常见，因为它们的索引不会特别大，但是查询量却很大。

2、分布/分片：基于哈希编码或某个其他机制将单个索引分布到多个节点或分片上去。一台主节点将到来的查询分到各个分片上然后对各部分的搜索结果进行汇聚。

除了通过哈希编码来分割索引之外，另一种做法是在逻辑上将索引和查询分配到独立的节点上去，这往往是可以做到的。例如，在多语言搜索中，可以按照语言来对索引进行分割处理，这样某个节点处理英语查询和文档而另一个节点则处理西班牙语查询和文档，这种做法可能有效（并不一定总是有效）。

自然而然，硬件的提高可以带来很好的结果，但是只能在加速比上带来改进，对于相关度来说没有任何帮助。下面考察一些久经考验的能够加速搜索并提高搜索质量的技巧和技术，即使它们带来的提高不容易获得。很显然，这样的考察也是很有意义的。

3.7.2 分析的改进

所有的搜索引擎，不管是闭源还是开源的，必须要定义一种机制来将输入文本转换成可以索引的词条。例如，Solr通过Analyzer过程来实现这一点，在Analyzer过程中，InputStream被分割成初始词条集合，然后对该集合进行修改（这点可选择）。这个分析过程就为索引在速度和相关度意义上的构建好坏设定了台阶。表3-7包含了本章前面部分提供的一些常见分析技术的一个重新列表，也包括一些新的技术，并且增加了一个如何起作用、有时会损害性能的注释。

表 3-7 常见的提高性能的分析技术

名 称	描 述	优 点	缺 点
去除停用词	在索引之前删除一些常见词，如 the、a、an 等，可以节省索引空间	更快构建且索引规模更小	是一种有损过程。最好索引停用词然后在查询时对它们进行处理。为了更好地进行短语搜索，停用词往往有用
词干还原	词干还原工具对词条进行分析，可能将它们转换成词根形式。比如，banks 转换为 bank	提高召回率	是一种有损过程，能够限制精确匹配搜索的能力。解决办法：保留两个字段，一个做了词干还原处理，一个没做
同义词扩展	对每个词条，增加 0 个或多个同义词。往往在查询时进行	通过返回不包含查询关键词但仍然与查询相关的文档提高召回率	有歧义的关键词可能会返回不相关文档
对词条进行小写处理	所有词条都转换成小写	用户常常输入大小写不正确的查询，在查询和索引时将它们都转换写小写能够获得更多的匹配	会阻碍大小写敏感的匹配。很多系统会保留两个字段，一个用于精确匹配，另一个用于非精确匹配
引入外部资源	有些查阅到的外部资源能够为词条的重要性提供额外信息，该重要性会编码为索引中词条所带的某种负载数据。例如前端权重、链接分析及词性	通常对于特定词条来说能够容许存储更多的意思，这些信息能够增强搜索的效果。例如，链接分析就是 Google PageRank 算法的核心（Brin 1998）	可能会显著降低分析的速度并增加索引的规模

另一种能够在困难情况下有助于提高结果但又不是特别常见的技术称为 n 元组分析，它实际上是我们在本书前面介绍的序列建模的一种形式。所谓 n 元组，指的是一个或多个连续字符或词条组成的序列。例如，对于 *example* 这个词基于字符的 1 元组（unigram）为：e、x、a、m、p、l、e，而其二元组（bigram）为 *ex*、*xa*、*am*、*mp*、*pl*、*le*。同样，基于词条的 n 元组会产生伪短语。例如，*President of the United States* 中的二元组有 *President of*、*of the*、*the United*、*United States*。为什么 n 元组非常有用？

它们主要源自近似匹配的场景需求，或者当数据不是特别清晰（如OCR数据）的情况。在Lucene中，拼写检查模块使用了 n 元组来产生候选建议结果，之后这些结果会评分排序。当搜索像中文一样的语言时，通过算法来确定词条往往很难，此时 n 元组就可以用于构建多种词条。这种方法不需要任何语言知识（Nie 2000）。基于单词的 n 元组在带停用词搜索是十分有用。例如，假定一篇文档包含如下两个句子：

- John Doe is the new Elbonian president.
- The United States has sent an ambassador to the country.

然后，这篇文档在使用停用词分析后会转换为这些词条：*john*、*doe*、*new*、*elbonian*、*president*、*united*、*states*、*sent*、*ambassador*、*country*。于是，如果输入查询是President of the United States，分析之后，查询转换为*president united states*，由于这三个词在上面这篇文档中都出现，因此该查询就会与文档相匹配。但是如果在索引阶段保留停用词，但是只用停用词在查询端来产生类短语 n 元组，由于产生像*President of the*和*the United States*之类的查询（也可以是其他形式的结果，主要取决于所产生的 n 元组类型），更可能匹配那些包含精确短语的句子，所以就会降低错误匹配的机会。在某些情况下，产生多种元组十分有用。再回到刚才那个例子，你可以产生2元组直到5元组，从而更可能精确匹配*President of the United States*。当然，这种技术并不完美，还会带来问题，但是它可以有助于使用停用词中的信息，否则这些信息就丢失了。

除了表3-7给出的技术和 n 元组之外，每个应用还有自己的需求。这种情况使开源方法的一个主要优点体现得更加明显，即源代码可以扩展。需要记住的是，分析涉及越多，索引构建也就越慢。由于索引构建通常是一项离线任务，如果能够获得可观收益的话，那么可能就值得做更多复杂的分析。但是一般的规则是，开始比较简单，某些特征能够解决问题的话，就加入这些特征。

解决了分析问题之后，下面看看如何提高查询性能。

3.7.3 提高查询性能

在查询这方面，有很多技术可以用于同时提高搜索速度和精度。在大部分情况下，提供好结果的难点在于用户信息需求的描述不足。这是人的本性使然。Google提

供的简单界面鼓励用户输入单关键词或双关键词查询，这显著地提高了Web搜索引擎及更小型搜索引擎的门槛。尽管大的互联网引擎能够访问Google一样的资源，但是小系统往往没法访问大量查询日志或像HTML链接一样的文档结构，也没法使用其他用户相关反馈机制来提供对用户而言很有价值的信息。花时间构建复杂方案之前，我们给出有助于改进结果的两个关键事项。

1. 用户训练：有时需要给用户展示出，通过学习一些关键语法技巧（如短语等）可以将检索结果提高到何种程度。

2. 外部知识：是否存在某个指示信息使得一篇或多篇文档比其他文档更重要？例如，也许该文档是CEO写的，或者100个人中有99人将其标为有用，或者该文档的边缘收益是对比文档的五倍。不管是什么，都要想办法将这个知识编码到系统当中并作为搜索的一个因素。如果搜索系统不容许这么做，那么可能是时候构建一个新系统了！

除了用户训练和使用索引的先验知识之外，还有很多办法可以提高查询速度和精度。首先，在大部分情况下，查询词项之间应该是AND而非OR的关系。例如，用户输入的是 *Jumping Jack Flash*，那么假设不是搜索短语的话，该查询应该转换成的等价形式为 *Jumping AND Jack AND Flash* 而不是 *Jumping OR Jack OR Flash*。通过使用AND，所有的查询词项都应该匹配。当然这种做法几乎可以肯定会提高正确率，但是可能会降低召回率。由于采用这种做法只需要对更少的文档进行评分，因此其速度肯定会更快。使用AND可能会导致零结果查询，但是如果想要结果的话之后可以回退到一个OR查询。对于简单查询AND可能不会产生足够的结果的一个唯一可能是文档集非常小（大概来讲，少于200000篇文档）。

注意 这里的AND使用并不意味着所有的搜索引擎都支持这种语法，但是Solr使用这种语法，因此为简单解释起见我们就保留这样的描述。

另一种能够带来大幅度正确率提升的查询技术是检测查询中的短语或者自动采用n元组词条产生短语。在前一种情况下，通过分析查询来确定查询中是否包含短语，之后这些短语转换成系统的内部短语查询表示。例如，如果用户的输入是 Wayne Gretzky career stats，那么一个好的短语检测器会将 Wayne Gretzky 识别为一个短语，然后生成查询 “Wayne Gretzky” career stats 或者甚至 “Wayne Gretzky career

stats”。很多搜索系统在指定短语时也提供基于位置的 $slop$ 因子。该因子指定了两个或多个单词位置之间相距的距离，在该距离内这些词也算构成短语。通常来说，单词之间距离越近，最后的得分也越高。

至于查询速度的提高，通常查询词项越少，搜索的速度也越快。同样，由于常见词会造成寻找相关文档时对大量文档评分，从而降低查询的处理速度，因此在查询时去除停用词是值得的。很显然，鼓励用户在提交查询词尽量避免使用歧义词或常见词也有助于查询速度的提高，但是如果用户大部分都不是专家用户的话，这种要求不太现实。

最后，一项常见的提高搜索质量而非搜索速度的技术会涉及每次用户输入至少两个查询的提交，这种技术称为相关反馈。相关反馈将一个或多个结果手工或自动标记为相关结果，然后利用其中重要的词项来构建新查询。在手工相关反馈中，用户会指出一篇或多篇文档相关（如通过选中复选框或点击某条链接），然后这些文档中的重要词项会用于构建新查询，之后该查询会自动提交给系统并返回新结果。在自动相关反馈中，返回的前5或10篇文档被自动认为相关，然后基于这些文档构建新查询。两种情况下，也可以识别哪些文档不相关，并将其中的词项从查询中剔除或者降低它们的权重。在很多情况下，保留下来的新查询词项和原始查询的权重计算不太一样，可以通过输入参数来指定原始查询词项或新查询词项的权重。例如，可以决定新词项的重要性是原始查询词项的两倍，这样就可以将新词项的权重乘以2。下面通过一个简单的例子来看看这种相关反馈的过程。假定所用文档集包含4篇文档，如表3-8所示。

表 3-8 一个文档集的例子

文档 ID	词 项
0	minnesota, vikings, dome, football, sports, minneapolis, st. paul
1	dome, twins, baseball, sports
2	gophers, football, university
3	wild, st. paul, hockey, sports

现在假设用户对Minnersota的体育项目感兴趣，于是他们会输入查询 minnesota AND sports。这是一个完全合理的查询，但是对于我们而言这个例子太小，在没有相关反馈的情况下只有文档0会被返回。但是如果使用自动相关反馈并使用排名第一的结果进行查询扩展，系统就会构建一个新查询，比如 (minnesota AND sports) OR (vikings OR dome OR football OR minneapolis OR st. paul) *2。这条新的反馈查询会导致所有的文档都会被返回（多么方便！）。不言而喻，相关反馈很少会带来这么好的结果，但是它确实能有所帮助，特别是在用户会提供判定结果的情况下尤其如此。要学习更多的有关相关反馈的知识，请参考*Modern Information Retrieval*（Baeza-Yates 2011）或*Information Retrieval: Algorithms and Heuristics*（Grossman 1998）。接下来，我们快速浏览一下其他评分模型来了解一些其他的搜索方法。

3.7.4 其他评分模型

前面我们主要关注了向量空间模型的评分方法，特别是Lucene中的模型，但是VSM中还有很多评分方式，除VSM之外还有很多其他评分模型，如果我们不提的话，那么就是我们的责任了。这些评分方法和评分模型的一部分如表3-9所示。这些方法中的大部分都在研究系统中实现或者受专利保护，对于开源生产用户而言不可访问或者不太实际。但是现在有一些方法或模型已经在Lucene内部实现并可以在Solr中进行配置。实际上，大部分Lucene的评分功能在未来的Lucene（或者取决于这本书的出版时间）和Solr 4.0中都是可插入的。由于我们现在用的是早期版本，这里就不对这些新模型的使用进行讨论。一种提高查询性能的方法就是对底层的评分模型进行切换。

表 3-9 其他的评分方法和模型

名 称	描 述
语言建模	另一种概率模型，它以另外一个角度来考虑 IR 问题，有点像 Jeopardy TV 秀节目，给出答案，竞争者必须提出问题。和度量给定文档是否和查询匹配不同，该模型会度量给定文档下建立查询的可能性
隐性语义索引（LSI）	一种基于矩阵的方法，它试图通过利用奇异值分解在概念层次上对文档建模。该方法受专利保护，因此开源用户很少使用

续表

名 称	描 述
神经网络及其他机器学习方法	在这些方法中，通过某个训练集和用户给出的反馈，会随时间推移学习到检索功能
其他权重机制	很多研究人员提出了在多种模型中使用的改进权重机制。有些改进方法的核心在于使用文档长度和平均文档长度作为评分因子（Okapi BM25、回转文档长度归一化等）。其基本的假设是，相对于短文档而言，长文档中关键词的 TF 值也越大，但是这些关键词重复带来的收益会随着文档变长而减小。注意，长度归一化因子往往不是线性的
概率模型及相关技术	利用统计分析方法来确定给定文档和查询匹配的似然。相关技术包括推理网络和语言建模

最后，注意到很多有关查询性能提高的研究都在进行中，通过不同的模型或不同的查询构建方法可以实现这一点。很多优秀的有趣的研究在SIGIR年会上发表。对这些论文进行跟踪对于了解最新的性能提高技术是一种很好的方法。无论如何，这里给出的信息已经足够你起步。接下来我们看看Solr性能相关的技术，以便能给你的系统带来具体的性能提升。

3.7.5 提升Solr性能的技术

尽管Solr高度可用，也存在很多需要遵循的最佳实践方法来了解Solr如何有机会实现性能的提升。为正当解决性能问题，我们将这个问题分成两个子问题，第一个子问题涉及索引性能，第二个子问题涉及搜索性能。在开始之前，通常最简单的提升性能的方法就是升级到最新版本。Solr的社区十分活跃，随时都有很多新的改进。

提升索引性能

索引性能问题可以分成三部分：

- Schema设计
- 配置
- 提交方法

如前所述，良好的Schema设计会估计需要哪些字段，它们是如何分析的，它们是否需要存储。搜索很多Field会比搜索一个Field要慢。同样，返回大量包含存储Field

的文档会比包含较少Field的文档要慢。此外，复杂的分析过程对索引性能具有负面影响，这是因为需要花费时间来执行复杂的切词和词条过滤过程。通常情况下，你可以牺牲一些质量来获取更高的速度，当然这取决于你的用户。

Solr配置也提供了很多控制性能的杠杆，其中包括通知Lucene（Solr底层的强大搜索库）如何创建并撰写索引存储文件。这些因素都会在solrconfig.xml的<indexDefaults>选项中指定，下面给出了一个示例：

使用 Lucene 的复合文件格式来存储文件描述符，此时所付出的代价是降低索引和搜索的速度

maxBufferedDocs 控制了写到硬盘之前内部缓冲文档的数目。该数目越大需要越多的内存，也会加快索引的速度，而该数目小意味着需要更少的内存

```
<useCompoundFile>false</useCompoundFile>

<mergeFactor>10</mergeFactor>

<maxBufferedDocs>1000</maxBufferedDocs>

<maxMergeDocs>2147483647</maxMergeDocs>

<maxFieldLength>10000</maxFieldLength>
```

mergeFactor 控制了 Lucene 合并内部文件的频繁程度。小数字（<10）会比默认值使用更少的内存，但付出的代价就是索引较慢。在大部分情况下，默认值已经足够，但是你可能想实验一下

maxMergeDocs 指定了 Lucene 可以合并的最大文档数目。小数目（<10000）更适合于索引更新频繁的系统，而大数目更适合于批索引，并且会加快搜索速度

maxFieldLength 指定的是单个 Field 中建立索引的最大词条数目。如果你预期拥有包含大量词条的大文档，那么就增大该值（和你的 Java 堆大小）

最后，应用程序如何将文档发送给Solr将显著影响索引的性能。最佳的实践经验建议，在使用HTTP POST机制时每次发送多篇文档。通过使用多限制来发送多文档请求、增加吞吐量、最小化HTTP开销等措施，可以进一步提升性能。Solr会关注所有的同步问题，所以放心，数据会被正确地索引。下面一节会列出上述提到的与性能有关的各种因素类型并描述其与Solr性能的关系。

搜索性能

搜索性能可以分成多种不同类型，每种类型提供了不同的性能水平（参考表 3-10）。

表 3-10 搜索性能类型

类 型	描 述
查询类型	Solr 支持丰富的查询语言，允许简单的词项查询到通配及范围查询。使用通配符和搜索范围的复杂查询将会比简单查询执行要慢
规模	查询大小（子句的数目）和索引规模对性能都起重要作用。索引越大，需要搜索的词项也越多（通常词项个数与文档数目呈亚线性关系）。更多的查询词项也往往意味着需要检查更多的文档和 Field。如果查询访问所有 Field 的话，更多 Field 也意味着需要检查更多的内容

续表

类 型	描 述
分析	和索引一样，复杂的分析过程会比简单分析过程要慢，但是这一点往往可以忽略不计，除非碰到真正很长的查询或者进行同义词扩充或查询扩展
缓存和预热策略	Solr 对于缓存查询、文档以及其他重要结构拥有一些高级机制。此外，Solr 能够在新索引可供搜索之前自动填充这些结构。有关缓存的信息可以参考 solrxonfig.xml 文件。查询日志分析和 Solr 管理界面可能帮助确定缓存办法在搜索中是否有用。如果没用（缓存命中率很低），最好关掉这个选项
复制	查询量大这个问题可以通过将 Solr 索引复制到多个负载均衡的服务器从而将查询分布到多台机器来解决。Solr 提供一套在多台服务器之间保持索引同步的工具
分布式搜索	大型索引可以分割（分片）到多台机器上。主节点会将输入的查询广播到所有的分片上并对结果进行整理。通过和复制技术一起使用，可以建立大型的容错系统

最后，和大多数优化策略一样，某个策略可能对某个应用有效而对另一个应用却无效。前面给出的指导策略只是提供了一个使用Solr的一般法则，对于你自己的情况，实际测试以及对数据和服务器进行分析是得到最优策略的唯一途径。接下来，我们看看除了Java和其他语言版本的Solr之外其他的一些搜索工具。

3.8 其他搜索工具

对于开源来说，一件伟大的事情就是任何人都可以策划一个项目然后让该项目为其他人所用（当然，这也是缺点，因为很难知道各个开源产品孰好孰坏）。在你的产品中有很多开源搜索库可供使用，其中很多库的设计目标并不相同。有些试图最快，而有些希望方便于测试新搜索技术而更偏学术研究。

尽管本书所有作者都广泛使用Solr和Lucene并倾向于使用这种解决方案，但是表3-11仍然提供了一些可选方法或其他语言的实现。

表 3-11 其他搜索引擎

名 称	URL	特 性	许 可 证
Apache Lucene 及其变种	http://lucene.apache.org	底层搜索库，需要更多工程实现，但是也提供了更多的灵活性通过对占用情况，内存等进行更多的控制。类似地，存在为其他语言调用的 Lucene API 实现，包括 .NET、Python (PyLucene) 和 Ruby (Ferret) 等等，每种实现都与 Java 版 Lucene 保持一定程度的兼容性	Apache 软件许可证 (ASL)
Apache Nutch	http://lucene.apache.org/nutch/	在 Apache Hadoop 和 Lucene Java 版本之上包含采集器、索引器和搜索引擎在内的全套服务	Apache 软件许可证 (ASL)
ElasticSearch	http://elasticsearch.com	一个基于 Lucene 的搜索服务器	Apache 软件许可证 (ASL)
Minion	https://minion.dev.java.net/	一个来自 Sun 公司的开源搜索引擎	GPL v2.0
Sphinx	http://www.sphinxsearch.com/	关注对 SQL 数据库中内容进行索引的搜索引擎	GNU 公共许可证 v2 (GPL)
Lemur	http://www.lemurproject.org/	使用另一种称为语言建模的排序公式来替代向量空间模型	BSD
MG4J—Managing Gigabytes for Java	http://mg4j.dsi.unimi.it/	优秀书籍 <i>Managing Gigabytes</i> (Witten 1999) 的配套搜索引擎。目标是快速可扩展。也提供了不同的排序算法	GPL
Zettair	http://www.seg.rmit.edu.au/zettair/	设计目标为快速紧凑，允许对 HTML 和 TREC 数据集进行索引和搜索	BSD

尽管存在大量搜索引擎，表3-11只提供了一部分不同语言版本不同许可证版本的优秀工具。

3.9 小结

搜索内容既丰富多彩，又复杂多样。能够更好地访问内容是对你生活中的大量数据进行控制的第一步。此外，搜索现在已经成了几乎所有面向顾客的应用的必要组成部分。像Amazon、Google、Yahoo!等公司已经展示了优秀搜索给用户和公司带来的机会。现在你需要利用本章的思想来进一步丰富你的应用。具体地，你应该了解内容索引和搜索的基本知识，如何搭建和使用Apache Solr以及如何使搜索（特别是Solr）在实际应用运行所涉及的相关问题。从这里出发，我们接下来考察结果并不总是确定性的文本情况，或者我们称之为模糊字符串匹配。

3.10 相关资源

Baeza-Yates, Ricardo and Ribiero-Neto, Berthier. 2011 Modern Information Retrieval: The Concepts and Technology Behind Search, Second Edition. Addison-Wesley.

Brin, Sergey, and Lawrence Page. 1998. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." <http://infolab.stanford.edu/~backrub/google.html>.

Grossman, David A. and Frieder, Ophir. 1998. Information Retrieval: Algorithms and Heuristics. Springer.

Nie, Jian-yun; Gao, Jiangfeng; Zhang, Jian; Zhou, Ming. 2000. "On the Use of Words and N-grams for Chinese Information Retrieval." Fifth International Workshop on Information Retrieval with Asian Languages, IRAL2000, Hong Kong, pp 141-148.

Salton, G; Wong, A; Yang, C. S. 1975. "A Vector Space Model for Automatic Indexing." Communications of the ACM, Vol 18, Number 11. Cornell University. http://www.cs.uiuc.edu/class/fa05/cs511/Spring05/other_papers/p613-salton.pdf.

"Vector space model." Wikipedia. http://en.wikipedia.org/wiki/Vector_space_model.

Witten, Ian; Moffatt, Alistair; Bell, Timothy. 1999. Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann, New York.

第4章 模糊字符串匹配

本章内容

- 利用前缀和 n 元组进行模糊字符串匹配
- 前缀匹配用于查询自动补齐
- n 元组及字符串匹配在查询拼写检查中的应用
- 字符串匹配在记录匹配任务中的应用

文本处理的一个最难之处在于很多任务本质上具有模糊性。不论是搜索结果的相关性还是对相似对象聚类，相关性或相似性的精确含义很难采用既直观又具体的方式表达出来。在语言中我们会随时碰到这种现象，通常我们不会对此有所考虑。例如，你可能会听到有关一个新乐队的描述如下：“他们像Radionhead乐队，只是有一点不同”。通常情况下你只会点点头，某种解释就会出现在脑海中，不会考虑其他有大范围可能的潜在意义和有效解释。

我们无疑会想起Google搜索中的Did You Mean功能（参考图4-1，当Google有足够确信度认为输入查询可能存在拼写错误时会用正确查询的返回结果替代）。尽管这个功能的目标用户是查询可能有拼写错误的用户，该功能对于拼写存在困难的用户来说无疑是一个福音。这不仅意味着你的搜索生产力得到了提升，也提供了一种方法来查找那些要么不在词典中出现，要么尝试输入和正确拼写足够接近的查询以产生合理的推荐。现在，你可以快速书写“joie de vivre of coding late into the evening”而不只是“coding is something to do”，这是因为现在查找借自于法语的短

语的正确拼写已经不是问题。

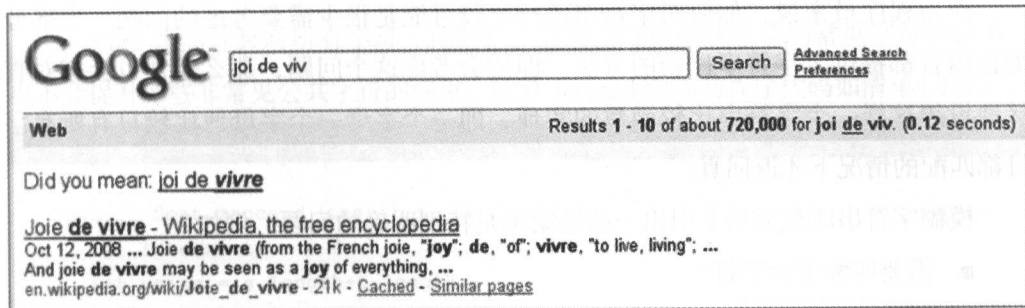


图4-1 Google搜索的Did You Mean样例，截图来自2009年2月1日

像Did You Mean或者有时称为查询拼写检查的功能需要模糊匹配。具体地，你需要为输入查询生成多种可能的推荐结果，并对它们进行排序，以确定是否需要将推荐结果展示给用户。模糊字符串匹配或就叫做模糊匹配，是指寻找相似但不一定精确匹配的字符串的过程。它有点像正则表达式匹配，但与后者稍微不同。与之相对，字符串匹配主要关注精确匹配。模糊匹配中的相似度往往通过距离、得分或者相似度似然来定义。例如，利用后面我们会介绍的编辑距离（Levenshtein距离）概念，可以得到*there*和*here*之间的编辑距离为1。尽管你可能会猜到这个具体得分的含义，但是我们现在还是暂时不讨论这个问题，在本章后面我们会再回到这个问题。

拼写检查仅仅是模糊字符串匹配的一个例子。另一个常见的例子往往出于公司合并时合并顾客表或者政府和航空公司检查乘客名单来发现潜在犯罪份子的需要。这些情况下实现的任务往往称为记录链接或实体消解，必须要将一份姓名表和另一份姓名表进行比较来确定两个拼写极其类似的客户是不是同一个人。直接简单地将姓名匹配并不能完全解决问题，但是这样做非常有用。为探讨这些使用案例，本章将考察多种模糊字符串匹配的方法，并考察这些方法的多个开源实现。你会学到如何对单词、短语和姓名进行比较，以识别其他与其相似的单词和短语并将这些单词和短语按照相似度排序。我们会考察如何在具体应用中应用这些技术并利用Solr和相对很少的定制Java代码来构建这些应用。最后，我们会将学到的技术进行组合以构建一些常见的模糊字符串匹配应用。

4.1 模糊字符串匹配方法

作为程序员来说，如何对字符串进行比较可能是很少需要考虑的问题。大部分编程语言都提供了字符串比较的方法。即使会考虑这个问题，那么通常也会相对直接地设想这样一个字符串比较函数的实现，即一个字母一个字母地比较只有所有字母都匹配的情况下才返回真。

模糊字符串匹配会马上引出一些答案不是特别明确的问题。例如：

- 需要匹配多少字符？
- 如果字母都一样而只是次序不一样呢？
- 如果有额外字母呢？
- 是不是有些字母比其他字母更重要？

不同的模糊串匹配方法对上述问题的解答也不相同。有些方法将字符的重合度作为主要的字符串相似度考察对象。其他一些方法对字符出现的次序更直接地建模，还有其他一些方法同时考察多个字母。我们将分成三小节来描述这些方法。第一节的标题为“字符重合度量方法”，在这一节中会考察Jaccard距离及其变形，还有介绍解决字符重合度量度的Jaro-Winkler距离。第二节的标题为“编辑距离度量方法”，将考察基于字符次序的方法及一些变形方法。最后，我们会在“ n 元组编辑距离”那一节介绍同时考察多个字母的方法。

4.1.1 字符重合度量方法

考察模糊串匹配的一种方法是字符重合度。直观上看，包含很多公共字符的两个字符串会比公共字符很少甚至没有的两个字符串更相似。本节将考察主要基于字符重合度的两种方法。第一种是Jaccard距离计算方法。我们首先会考察这种距离计算方法及其变形，然后考察Jaro-Winkler距离。

Jaccard距离

Jaccard距离，或称相似系数是一种实现“包含相同字符越多的两个字符串也越相似”这种直觉的方法。在用于字符串比较时，它是两个字符串共同包含的独立字符个数占两个串所有独立字符个数的百分比。更形式化地，设A是第一个字符串的字符集合，而B是第二个字符串的字符集合，那么Jaccard距离可以计算如下所示。

$$\frac{|A \cap B|}{|A \cup B|}$$

Jaccard 距离计算对所有字母一视同仁，它不会降低那些重合的常见字符的权重，也不会提升那些非常见公共字符的权重。计算 Jaccard 距离的程序代码如清单4-1所示。

清单4-1 Jaccard距离计算

```
public float jaccard (char[] s, char[] t) {
    int intersection = 0;
    int union = s.length+t.length;
    boolean[] sdup = new boolean[s.length];
    union -= findDuplicates (s, sdup);
    boolean[] tdup = new boolean[t.length];
    union -= findDuplicates (t, tdup);
    for (int si=0;si<s.length;si++) {
        if (!sdup[si]) {
            for (int ti=0;ti<t.length;ti++) {
                if (!tdup[ti]) {
                    if (s[si] == t[ti]) {
                        intersection++;
                        break;
                    }
                }
            }
        }
    }
    union-=intersection;
    return (float) intersection/union;
}

private int findDuplicates (char[] s, boolean[] sdup) {
    int ndup =0;
    for (int si=0;si<s.length;si++) {
        if (sdup[si]) {
            ndup++;
        }
        else {
            for (int si2=si+1;si2<s.length;si2++) {
                if (!sdup[si2]) {
                    sdup[si2] = s[si] == s[si2];
                }
            }
        }
    }
}
```

寻找重复字符并从并集中减去这些字符

跳过重复字符

寻找交集

返回 Jaccard 距离

```

    }
    return ndup;
}

```

在清单4-1所示的实现代码中，首先计算的是公式中的并集部分（分母），通过从一个记录并集字符个数的计数器减去重复字符个数即可得到该值。接下来计算的是两个字符串的公共元素数目（分子）。最后，分子除以分母返回最后得分。

Jaccard距离的一个常见扩展方法基于出现频率给每个字符赋以权重，权重可以采用如第3章介绍的TF-IDF值。利用这种加权的方法，余弦相似度就是一种很自然的在搜索字符串时计算字符串相似度的方法，但是它返回的值区间是-1到+1。为了将该结果归一化到0到1之间，我们可以按照下面的方式对原有计算方法稍加修改：

$$\frac{A \cdot B}{\|A\|^2 + \|B\|^2 - A \cdot B}$$

最终的计算公式称为谷本系数，当所有字符权重一样时，谷本系数就等价于Jaccard系数。

由于Solr在检索时使用的是基于余弦的相似度评分方法，它提供的相似性结果和谷本系数类似，因此最简单的实现这种评分机制的方法是在Solr中将每个词项看成文档将每个字符看成词条，然后对词典或其他词项集合建立索引。上述做法可以通过指定一个模式切词器来实现：

```

<fieldType name="characterDelimited" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.PatternTokenizerFactory" pattern="."
                                                    group="0" />
  </analyzer>
</fieldType>

```

然后就可以利用某个词项对该字段进行查询，而Solr将提供一个基于词典中字符出现频率的余弦词项排序方法。如果想尝试这样索引词项的一个快速版本的话，参考com.tamingtext.fuzzy.OverlapMeasures及其cosine方法。实际当中，这可能会产生合理的结果，但是利用相似度得分可能很难在结果当中进行抉择。在评分中，该方法也没有考虑字符的位置信息，而该信息有助于更好地选择推荐结果。为解决上述担忧，接下来介绍能够考虑位置信息的另一种距离计算方法。

JARO-WINKLER距离

基于字符重合度方法的一个缺点在于它们没有对字符的出现次序建模。例如，在前面介绍的距离计算方法当中，字符串开头的一个字符可能和另一个字符串结尾的一个字符匹配上，这种匹配和在字符串近似区域产生的匹配一样对待。这种情况的一个极端版本是，如果某个单词反转了，那么它和精确匹配上的字符串得分一模一样。Jaro-Winkler距离试图通过三种途径启发式地处理这个问题。首先是按照长字符串的长度来限制，必须要在第二个字符串的某个窗口内进行字符匹配。其次不以相同次序出现的置换或匹配字母的数目也要考虑在内。最后，基于两个单词最长公共前缀的长度来增加一个奖励值。很多有关该方法的有趣讨论可以在互联网上找到，如果对该距离感兴趣的话，一个高效的实现已经内置于Lucene的org.apache.lucene.search.spell.JaroWinklerDistance类中。

基于字符重合度的方法的主要缺点是没有很好地对字符次序建模。下一节中，我们会介绍一种称为编辑距离的方法，它能够更正式地对字符次序建模。对字符次序建模通常会比仅使用字符重合度带来更高的计算开销。我们也会介绍一些高效计算的方法。

4.1.2 编辑距离

另一种确定两个字符串之间相似度的方法是编辑距离。所谓编辑距离指的是一个字符串转换为另一个字符串所必须的编辑操作次数。编辑距离有多种不同形式，但是通常包括插入、删除和替换操作：

- 插入操作会在源字符串中插入一个字符来使其与目标字符串更相似。
- 删除操作会去掉一个字符。
- 替换操作会用目标字符串的一个字符替换源字符串的一个字符。

编辑距离是从一个字符串转换到另一个字符串所必须的插入、删除和替换操作次数之和。例如，要将*tamming test*转换成*taming text*需要删除一个*m*并将*s*替换为*x*，这样得到的编辑距离就是2。这种允许插入、删除和替换操作并赋予每种操作相同权重1的简单编辑距离形式称为Levenshtein距离。

计算编辑距离

尽管存在多种操作序列可以将一个字符串转换为另一个字符串，通常希望两个

字符串的编辑距离是实现这一目标的最少操作数目。计算从一个字符串到另一个字符串所需的最少操作次数初看起来计算开销很大，但是实际上可以通过 $n*m$ 次比较就可以实现，其中 n 、 m 分别是两个字符串的长度。实现这个任务的算法是一个典型的动态规划算法，其中整个问题被分解成给定偏移后确定两个字符串前缀的最优编辑操作序列的子问题。程序清单4-2中的代码就是Java中的缩减版本。记住，我们选择了十分直接的方法来实现Levenshtein距离，也存在更加高效的使用更少内存的实现方法。关于这一点我们留给读者自己练习。

清单4-2 编辑距离计算

```
public int levenshteinDistance (char s[], char t[]) {
    int m = s.length;
    int n = t.length;
    int d[][] = new int[m+1][n+1];

    for (int i=0; i<=m; i++)
        d[i][0] = i;
    for (int j=0; j<=n; j++)
        d[0][j] = j;
    for (int j=1; j<=n; j++) {
        for (int i=1; i<=m; i++) {
            if (s[i-1] == t[j-1]) {
                d[i][j] = d[i-1][j-1];
            } else {
                d[i][j] = Math.min (Math.min (
                    d[i-1][j] + 1,
                    d[i][j-1] + 1),
                    d[i-1][j-1] + 1);
            }
        }
    }
    return d[m][n];
}
```

分配距离矩阵

初始化距离上界

代价和前一个匹配一样

插入删除和替换的代价都是 1

表4-1给出了两个示例字符串taming text和 tamming test之间的距离矩阵。每个元素包含的是从一个字符串转换另一个字符串的最小编辑代价。例如，可以在第3行4列看到 m 的删除代价，在第10行第11列看到 s 替换 x 的代价。最小的编辑距离往往出现在距离矩阵的右下角。

表 4-1 计算编辑距离的矩阵

		t	a	m	m	l	n	g		t	e	s	t
	0	1	2	3	4	5	6	7	8	9	10	11	12
t	1	0	1	2	3	4	5	6	7	7	8	9	10
a	2	1	0	1	2	3	4	5	6	7	8	9	10
m	3	2	1	0	1	2	3	4	5	6	7	8	9
i	4	3	2	1	1	1	2	3	4	5	6	7	8
n	5	4	3	2	2	2	1	2	3	4	5	6	7
g	6	5	4	3	3	3	2	1	2	3	4	5	6
	7	6	5	4	4	4	3	2	1	2	3	4	5
t	8	6	6	5	5	5	4	3	2	1	2	3	4
e	9	7	7	6	6	6	5	4	3	2	1	2	3
x	10	8	8	7	7	7	6	5	4	3	2	2	3
t	11	8	9	8	8	8	7	6	5	3	3	3	2

现在已经能够计算编辑距离了，接下来看看如何使用它。

归一化编辑距离

在大多数使用编辑距离的应用中，会希望设置一个编辑距离阈值以排除那些很不相似的校正结果，或者换句话说，那些需要太多编辑操作的校正结果。通过运行上述计算方法很快会就遇到一个问题。直观上看，同是编辑距离2，相对于长度为10的字符串而言，长度为4的字符串所需的编辑相对于长度要多得多。此外，对于某个字符串，必须要基于编辑距离对多种可能的校正结果字符串进行排序。记住，这些校正结果字符串可能长度不同。为了比较不同长度情况下的编辑距离，基于字符串长度来对编辑距离做归一化处理十分有用。

为将编辑距离归一化到0至1之间，可以将两个字符串中较长字符串的长度减去

编辑距离然后再除以这个长度。在前面讨论的例子中，较长的字符串应该是tammimg test，长度为12，于是12减去2再除以12，得到0.833。如果要纠正的是较短的字符串，比如将tammimg编辑成taming，此时的归一化编辑距离就是6减去2再除以6，结果是0.666。上述归一化处理之后得到的结果和直觉是吻合的，即第二个例子中编辑距离的改变幅度比第一个例子大。由于归一化之后不同长度之间的结果可比较，上述过程也使得距离阈值的设定更加容易。

对编辑操作加权

对于不同的应用，确定编辑距离的编辑操作可以加权。这种情况下，编辑距离就是所有在字符串间转换的每个操作的权重之和。正如你在前面看到的Levenshtein距离一样，最简单的权重机制给每个操作赋予的权重为1。对不同操作赋予不同权重对于不同操作并不同等重要场合十分有用。例如，在拼写纠错中，两个元音字母之间的替换会比两个辅音字母之间的替换更有可能发生。对不同操作赋予不同的权重或者基于操作对象采用不同的操作有助于更好地捕捉上述区别。

Levenshtein距离的一个常见变形是Damerau-Levenshtein距离。该计算方法允许一个额外的相邻字母交换操作。可以认为该方法引入了另一种权重机制，在这种机制下相邻两个字母交换操作的权重为1，而在传统的机制下，需要插入和删除两个编辑操作才能完成同样功能。

互联网上有大量关于Levenshtein距离的讨论，包括算法的形式化分析及正确性证明。对于使用该距离计算方法感兴趣的人而言，Lucene中的org.apache.lucene.search.spell.LevenshteinDistance类中内置了一个优化的实现版本。该版本中只需要分配空间来存储距离矩阵的两行，在下一行计算的时候只需要前一行即可。该版本中也采用了前面介绍的长度归一化处理方法。

4.1.3 n 元组编辑距离

在迄今为止考察的编辑距离的各种变形方法当中，所有的操作都只涉及单个字符。一种扩展编辑距离概念的方法是同时容许多个字符，该概念叫作 n 元组编辑距离。该距离采用了Levenshtein距离的思想，而只是将每个 n 元组看成一个字符。利用2元组考察前面讨论的那个例子，得到的距离矩阵如表4-2所示。

表 4-2 计算 n 元组编辑距离的矩阵

		ta	am	mm	mi	in	ng	g_	_t	te	es	st
0	1	2	3	4	5	6	7	8	9	10	11	
ta	1	0	1	2	3	4	5	6	7	8	9	9
am	2	1	0	1	2	3	4	5	6	7	8	9
mi	3	2	1	1	1	2	3	4	5	6	7	8
in	4	3	2	2	2	1	2	3	4	5	6	7
ng	5	4	3	3	3	2	1	2	3	4	5	6
g_	6	5	4	4	4	3	2	1	2	3	4	5
_t	7	6	5	5	5	4	3	2	1	2	3	4
te	8	7	6	6	6	5	4	3	2	1	2	3
ex	9	8	7	7	7	6	5	4	3	2	2	2
xt	10	9	8	8	8	7	6	5	4	3	2	3

这种 n 元组方法的影响是,不涉及双字母的插入和删除操作会比一元组方法获得更重的惩罚,而替换操作的惩罚是一样的。

n 元组编辑距离的增强

通常有多种增强措施应用于 n 元组方法。第一种是注意到起始字符只出现在单个 n 元组中,而中间字符往往参与到所有 n 元组。在很多应用中,这些起始字符在匹配时比中间字符更重要。利用这种思路的一种方法称为附缀法,是指在字符串开始附加 $n-1$ 个字符(不论 n 取什么值)。这样做的结果是,第一个字符和中间字符所出现的 n 元组数目相同。此外,不以相同 $n-1$ 个字符开始的单词会因为没有和前缀匹配而得到一定惩罚。相同的过程可以应用于字符串尾部,此时字符串尾部的匹配才被认为重要。

第二种增强措施是对于共享相同字符的 n 元组允许某种程度的部分得分。可以使用Levenshtein距离来确定两个 n 元组之间的编辑距离并通过除以 n 元组的长度来将该值归一化到0到1之间。采用简单的位置匹配来代替Levenshtein距离也行得通。在这种方法中,可以对两个 n 元组相同位置上的字符是否匹配进行计数。当 n 大于2时,这种方法比Levenshtein距离方法更快。当 $n=2$ 时,两种方法等价。表4-3给出了包括附缀法和

部分匹配部分得分的情况。

表 4-3 增强方式下计算 n 元组编辑距离的矩阵

		0t	ta	am	mm	mi	in	ng	g_	_t	te	es	st
0	1	2	3	4	5	6	7	8	9	10	11	12	
0t	1	0	1	2	3	4	5	6	7	8	9	10	11
ta	2	1	0	1	2	4	5	6	7	8	8.5	9.5	10.5
am	3	2	1	0	1.5	3	4	5	6	7	8	9	10
mi	4	3	2	1	0.5	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8
in	5	4	3	2	1.5	1.5	1.5	2.5	3.5	4.5	5.5	6.5	7
ng	6	5	4	3	2.5	2.5	2.5	1.5	2.5	3.5	4.5	5.5	6
g_	7	6	5	4	3.5	3.5	3.5	2.5	1.5	2.5	3.5	4.5	5
_t	8	6.5	6	5	4.5	4.5	4.5	3.5	2.5	1.5	2.5	3.5	4
te	9	7.5	6.5	6	5.5	5.5	5.5	4.5	3.5	2.5	1.5	2.5	3
ex	10	8.5	7.5	7	6.5	6.5	6.5	5.5	4.5	3.5	2.5	2	3
xt	11	9.0	8.5	8	7.5	7.5	7.5	6.5	5.5	4.5	3.5	3	2.5

Lucene中包含了使用附缀法和长度归一化的 n 元组编辑距离的一个实现。该实现位于org.apache.lucene.search.spell.NgramDistance类中。

本节考察了多种基于两个字符串的距离确定两者相似度的方法。首先考察了基于字符串重合度的方法，比如Jaccard距离以及基于字母权重加权的扩展计算方法。之后还讨论了一种利用字符串滑动窗口来计算重合度的Jaro-Winkler距离计算方法。接着介绍了编辑距离并考察了其中一种简单的形式，即Levenshtein距离。再接着讨论了在此基础上的一些增强措施，包括长度归一化和不同编辑操作的权重定制，最后将该计算方法扩展到多字母上。迄今为止，我们都假设需要比较两个字符串的距离并集中关注比较的实现方法。下一节将考察如何寻找在给定输入字符串下不需要计算它与所有候选串之间的距离就可以找到较好匹配串的方法。

4.2 寻找模糊匹配串

能够计算两个字符串之间的相似度很有用，但是前提是已经拥有了两个字符串。在很多使用字符串匹配的应用中，只有作为前面介绍的相似度计算函数的其中一个输入字符串。例如，在拼写纠正当中通常只有一个没在词典中出现的怀疑可能拼错的单词。如果有一系列推荐词表的话，就可以使用前面的函数来对这个词表中的词排序，并将排名靠前的单词推荐给用户。理论上说，可以计算输入单词和词典中所有单词的相似度来获得推荐。不幸的是，这种做法计算开销太大，并且大部分单词和输入单词的相似度很低，几乎没有太多公共字符。实际当中需要快速方法来确定一个较小的候选词集合以便在这个小集合上运行相对耗时的比较算法。本节先介绍两种确定候选词表的方法，分别是前缀匹配和 n 元组匹配，然后介绍它们的高效实现。

4.2.1 在Solr中使用前缀来匹配

一种快速确定与某个字符串相似的字符串集合的方法是前缀匹配。前缀匹配返回的是和待匹配字符串有公共前缀的字符串集合。例如，如果要纠正单词 *tamming*，只考虑前缀包含 *tam* 的单词会将一部近100000词的词典减少到35个词，这35个词是如下7个词的各种形式：*tam*、*tamale*、*tamarind*、*tambourine*、*tame*、*tamp*和*tampon*。从计算角度来看，这显著节省了开销，由于字符串共享了公共前缀，可以保证它们之间有共同之处。

一种实现前缀匹配的做法是使用Solr。当文档输入到Solr索引时，可以计算给定长度的所有前缀并将它们存储为Solr将匹配的项。查询处理时，可以对查询执行相同的操作，包含某种程度前缀匹配的项列表会返回。由于这是一项相对常见的任务，因此Solr包含了一个称为EdgeNGramTokenFilter（类的全名为org.apache.lucene.analysis.ingram.EdgeNGramTokenFilter）的实现。最终的结果是，当像*taming*一样的项被索引时，项*ta*、*tam*、*tami*和*tamin*也同时被索引。正如在第3章介绍的那样，这可通过在schema.xml文件中指定索引及查询时的文档字段类型来实现。从程序清单4-3中可以了解这种做法。

清单4-3 为Solr前缀匹配指定字段类型

```
<fieldtype name="qprefix" stored="false" indexed="true"
  class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EdgeNGramFilterFactory"
      side="front" minGramSize="2" maxGramSize="3"/>
  </analyzer>
</fieldtype>
```

在上例中，我们使用了Solr内置的EdgeNGramFilterFactory类、空白字符切词器以及小写词条过滤器来产生前缀。

该过滤器同时在索引和查询时使用。在查询时，它产生用于匹配索引时生成前缀的匹配前缀。这种功能常用语查询提前输入，也称为自动推荐。由于前缀匹配的意义容易理解，因此前缀匹配对于上述应用十分有用，用户会看到一些和查询前缀匹配的一系列单词，并能理解为什么这些单词会出现，也能理解输入其他字符如何会使列表内容发生变化。本章后面会探讨查询提前输入，下一节我们将考虑在内存中实现前缀高效存储的数据结构，对于一些应用来说该结构是必须的。

4.2.2 利用trie树进行前缀匹配

尽管Solr可以用于前缀匹配，有些情况下对每个待查找的前缀都连接一个Solr示例似乎不太现实。这种情况下最好能够在一个内存中的数据结构中执行前缀查询。一种十分符合这种任务要求的数据结构是trie（有些人发音为“tree”，有些人发音为“try”）。

TRIE的定义

Trie或者称为前缀树是一个将字符串分解为字符进行存储的数据结构。字符构成树中的路径，直到树的子节点唯一表示一个字符串或者该字符串中所有的字符都已使用为止。在图4-2所示的trie树中，可以看到大部分单词都通过只代表其部分字符节点来表示，因为它们是具有特定前缀的独有单词。但是单词tamp在trie树中也是另一个词tampon的前缀，因此在trie中它的每个字符都有一个节点来表示。

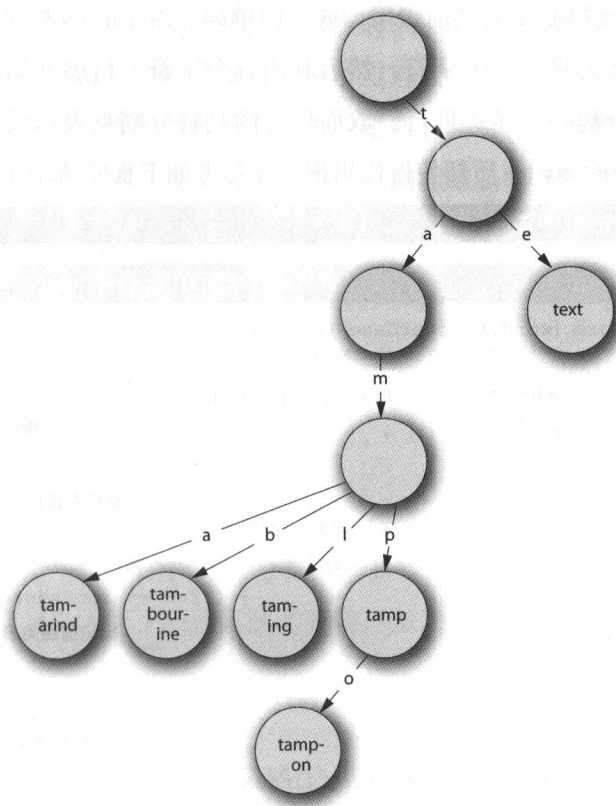


图4-2 一棵表示tamarind、tambourine、taming、tamp、tampon及text的trie树

TRIE树实现

Trie树的一个简单实现与其他任意树结构非常类似。在Trie树中，每个节点子节点的数目通常等于trie树所代表的字母表大小。从程序清单4-4中，可以看到一个只包含小写字母a-z的字符串的trie树实现。

清单 4-4 构建trie树节点

```

private boolean isWord;           ← 该前缀是否构成一个词
private TrieNode[] children;
private String suffix;           ← 如果前缀唯一，则表示词的剩余部分

public TrieNode (boolean word, String suffix) {
    this.isWord = word;
    if (suffix == null) children = new TrieNode[26]; ← 为每个字母初始化子树
    this.suffix = suffix;
}

```

Trie树必须要支持基于公共前缀增加或返回单词。由于trie是树结构，很常见的做法是利用递归法来去除第一个字符，然后利用剩余字符下行展开结构。基于性能原因考虑，字符串分割并非显式进行，取而代之的是将分割点表示成整数。尽管要考虑多种情况，本身的递归本质却使得代码很短（参考如下程序清单）。

清单4-5 在trie树中添加词

```

public boolean addWord (String word) {
    return addWord (word.toLowerCase (), 0) ;
}

private boolean addWord (String word, int index) {
    if (index == word.length ()) {
        if (isWord) {
            return false;
        }
        else {
            isWord = true;
            return true;
        }
    }

    if (suffix != null) {
        if (suffix.equals (word.substring (index))) {
            return false;
        }
        String tmp = suffix;
        this.suffix = null;
        children = new TrieNode[26];
        addWord (tmp, 0) ;
    }

    int ci = word.charAt (index) - (int) 'a';
    TrieNode child = children[ci];
    if (child == null) {
        if (word.length () == index + 1) {
            children[ci] = new TrieNode (true, null) ;
        }
        else {
            children[ci] = new TrieNode (false, word.substring (index+1)) ;
        }
        return true;
    }

    return child.addWord (word, index+1) ;
}

```

检查是否到达词尾

词存在返回 false

标记前缀为词

检查该节点是否包含后缀

词存在返回 false

分割前缀

前缀及后缀
构建新单词

前缀构建新单词

在下一字符上递归

检索单词时，可以通过遍历树结构找到代表所查询前缀的节点来完成。通过浏览前缀的每一个字符并访问该字符的子节点可以实现这一点。当前缀节点找到以后，可以运行一个深度优先的搜索算法来收集所有具有公共前缀的词。在前缀节点不存在的情况下，根据节点表示的单词和所查询前缀是否匹配，至多返回一个词。该方法的一个实现版本如程序清单4-6所示。

清单4-6 从trie树中检索单词

```
public String[] getWords (String prefix, int numWords) {
    List<String> words = new ArrayList<String> (numWords) ;
    TrieNode prefixRoot = this;
    for (int i=0;i<prefix.length () ;i++) {
        if (prefixRoot.suffix == null) {
            int ci = prefix.charAt (i) - (int) 'a';
            prefixRoot = prefixRoot.children[ci];
            if (prefixRoot == null) {
                break;
            }
        }
        else {
            if (prefixRoot.suffix.startsWith (prefix.substring (i))) {
                words.add (prefix.substring (0, i) +prefixRoot.suffix) ;
            }
            prefixRoot = null;
            break;
        }
    }
    if (prefixRoot != null) {
        prefixRoot.collectWords (words, numWords, prefix) ;
    }
    return words.toArray (new String[words.size () ] ) ;
}

private void collectWords (List<String> words,
                           int numWords, String prefix) {
    if (this.isWord ()) {
        words.add (prefix) ;
        if (words.size () == numWords) return;
    }
    if (suffix != null) {
        words.add (prefix+suffix) ;
        return;
    }
}
```

遍历树直到前缀消耗完

处理前缀没有被分割的情况

收集所有前缀节点子节点的词

```

    }
    for (int ci=0;ci<children.length;ci++) {
        String nextPrefix = prefix+ (char) (ci+ (int) 'a') ;
        if (children[ci] != null) {
            children[ci].collectWords (words, numWords, nextPrefix) ;
            if (words.size () == numWords) return;
        }
    }
}
}

```

上述trie树的实现对于添加和检索词十分高效，但是其表示需要字母表大小的数组。该数组会为每个不包含后缀的节点所构建。尽管这样做会使得字符词查找十分搞笑，但通常情况下可能下一个字母只有很少一部分真正存在。其他一些trie树的实现方法，如双数组trie树，会减少存储转换状态的内存消耗但在插入过程中会引入添加所需的消耗。Trie树中的很多使用场景如词典查找都基于相对静态的内容进行，由于添加单词时额外的工作只是一次性消耗，因此上述做法非常有用。有关该方法的一个讨论可以参见“An efficient digital search algorithm by using a double-array structure”（Aoe 1989）。

SOLR中的TRIE树

Solr 3.4中支持一个基于trie思想的数值型字段的实现，能够大幅度提高这些字段上的范围查询性能。这可以通过指定字段类型为trie来实现，如程序清单4-7所示。

清单4-7 使用Solr TrieField类型

```

<fieldType name="tint" class="solr.TrieField" type="integer"
    omitNorms="true" positionIncrementGap="0" indexed="true"
    stored="false" />

```

和前面给出的trie树实现不一样，Solr中实现的版本更像是早讨论的前缀词条方法。尽管trie树字段用于Solr中的数值类型，为了解其工作机理，首先考虑一个使用字符串的例子。如果想在*tami*到*tamp*之间执行一个范围查询，你可能会使用它们的公共前缀*tam*来限制你的搜索，然后来检查每篇返回文档字段是否在该范围之内。正如你在前缀情况下看到的那样，这可以显著减低与范围查询潜在匹配的文档数目。你可以通过引入前缀的增量概念进一步对搜索进行优化，比如*tamj*、*tamk*、*taml*...*tamp*，然后对返回的与这些前缀匹配的文档进行搜索。注意，任意与*tamp*前面的增

量前缀匹配的文档只包含与范围查询相匹配的文档。

为确定到底哪些文档匹配，你只能考虑那些与边缘前缀匹配的文档，并将它们与实际的范围查询进行比较。与增量前缀匹配文档的数目相关该前缀的长度，决定了计算文档和范围查询匹配所需要执行的比较次数。这里我们使用一个字符的增量应用与四个字符的字符串，但是其他还有多种可能的增量形式。可以设想在一个整数区间搜索，比如在[314 TO 345]区间搜索所有以前缀31、32、33和34开始的数字。Solr.TireField使用了相似的方法来基于二进制计算整数和浮点数的数值范围，这点与例中十进制的表示形成鲜明对照。

这一节考察了能够高效插入和检索前缀的trie树表示。我们给出了一个简单的实现，并讨论Solr中是如何利用类似思想来提高范围查询的性能的。下一节当中，我们不再仅仅限于前缀匹配，而是考察更稳定的不只是将单词首部字符考虑在内的匹配方法。

4.2.3 使用 n 元组进行匹配

尽管前缀匹配非常强大，但是它也有很多不足。其中一个不足是使用前缀匹配推荐的任意词项都必须包含公共前缀。对于一个首字符不对的单词或者词项，前缀匹配方法永远不能有这种纠正类型的推荐结果。尽管这种情况可能很少，但不是没听说过。接下来考察另外一个对于这种情况更鲁棒的技术。

在前面一节中可以看到，前缀可以用于限制对用户输入字符串进行匹配的字符串集合大小。也可以看到前缀更大也会候选推荐字符串的数目也会更少，但是同时最佳词项被排序在外的风险也会更大，这是因为纠正字符串必须包含前缀中的一个字符。当考虑大小为 n 的前缀就是字符串的第一个 n 元组时，一个扩展的使用前缀方式思路的方法被提出。通过同时考虑第2、3以及剩余的 n 元组，可以将前缀的概念推广到字符串的任意位置。

n 元组匹配方法将可能的匹配限制到与查询字符串共享一个或多个 n 元组的字符串上。利用前面tanning的例子，可以考虑包含tam的字符串，同时考虑该字符串的所有其他三元组：amm、mmi、min和ing。应用到前面那部100000万词的词典，只有近十分之一的单词能够与其中一个三元组匹配。尽管这看起来与单独的前缀方法降低的显著性差不多，但是现在可以处理原始文本中更大可能范围的错误，其中包括

首字母不正确的情况。该 n 元组方法也提供了对多种匹配进行排序的一个直观的方法，假如单词匹配了多个 n 元组，那么匹配数更多通常也意味该单词的推荐排名更高。在前面的例子中，有19个单词匹配上了5个 n 元组中的4个，74个单词（也包括刚才的19个单词）匹配上了4个 n 元组中的3个。通过对 n 元组的匹配结果排序，可以考虑那些仅仅考虑固定数目推荐结果的方法，以合理的确定性与最优最低编辑距离的单词在排名靠前的推荐结果之内。

Solr中的 n 元组匹配

和前缀匹配类似，Solr也通过org.apache.lucene.analysis.ngram.NGramTokenFilter及其关联工厂类org.apache.solr.analysis.NGramFilterFactory实现了 n 元组分析功能。

N 元组匹配没有捕获的信息是位置信息。某个 n 元组出现在字符串首部但是却与另一个字符串的尾部相匹配，对于这个 n 元组来说没有什么区别。一种克服这种限制的方法是使用字符串affixing方法来获取是在字符串首部还是尾部出现的位置信息。我们在4.1.3节已经介绍过这种做法。

本节主要讨论了快速查找模糊匹配字符串的技术。这些工具和前面计算编辑距离的技术相结合可以寻找并对模糊匹配串排序。下面将考察如何组合这些工具来构建应用。下一节我们会考察三个使用这些技术的应用。

4.3 构建模糊串匹配应用

基于前面介绍的工具，本节会考察模糊串匹配在实际应用的三种使用方法。具体地，我们会考察搜索中的提前输入功能，查询拼写检查以及记录匹配。提前输入功能通常也称为自动补齐或自动推荐，能够提供索引中的词项示例，通过选择单词而不是完成输入来减少键盘输入次数。另外，该功能还有一个好处就是用户知道单词的拼写是正确的，因而能有更好的搜索体验。对查询进行拼写检查往往出现在诸如Google网页的Did You Mean部分，它是一种引导用户采用另一种拼写并有更好搜索结果的单词的方法。注意，这里我们说的是另一种，因为它不一定是一个正确拼写的单词。某些情况下，索引中大部分的词项的拼写都不对（例如在线论坛），因此这种情况下可能展示非正确拼写的单词会带来更好的结果。最后，记录匹配，有时也称为记录链接或实体消解是确定两条不同记录实际描述同一对象的过程。例如，当合并两个用户数据库时，记录匹配过程就会确定某条记录中的Bob Smith是不是就

是另外一条记录中的Bob Smith。由于这三个案例给出了当前文本应用模糊串匹配的常见应用，因此接下来我们将集中关注这几个案例。

4.3.1 在搜索中加入提前输入功能

很多应用中的一个常见功能就是文本输入的自动完成。例如在编程时，很多集成开发环境（IDE）中会自动完成变量名的输入。在搜索应用中，当用户在搜索框中输入查询时往往使用提前输入功能，查询的可能完整词项会随着输入提示给用户。这个功能不仅可以节省用户输入整个查询的时间，通过仅仅推荐那些在搜索索引中会匹配一些文档的查询，它也可以指导搜索过程的优化过程。提前输入允许用户快速将其搜索转换为可以提供更好结果的短语，从而提高用户的总体体验。基于用户的输入提供推荐查询可以认为是将输入查询看成前缀的结果。对于前面的例子，可以使用Solr来返回这些查询的结果。

在Solr中索引前缀

第一步是允许Solr创建前缀查询。对于不完整的部分查询而言，可以再次使用EdgeNGramFilterFactory来计算前缀并将它们加入到生成的词条集合中。和前面一样，可以在schema.xml文件中指定一个字段类型然后分配一个字段来存储前缀（参考清单4-8）。记住在这种情况下，当用户输入时有一个最大的 n 元组大小来为更多变形负责。这会增加底层数据结构的规模，但是仍然是可控的。

清单4-8 在Solr中为提前输入指定字段类型

```
<fieldtype name="prefix" stored="false" indexed="true"
  class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EdgeNGramFilterFactory"
      minGramSize="2" maxGramSize="10"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldtype>
```

为索引指定分析器

使用边约束的 n 元组（前缀）

查询已是前缀，去除边 n 元组过滤器

该字段类型能够处理多词查询，允许从词边界开始的匹配。注意，EdgeNGramFilterFactory不必应用到查询上去，因为它已经是一个前缀。为应用该字段类型，必须要指定一个使用该类型的字段并索引植入该字段的文档。对于本例而言，假设用字段word来索引词典中单词。然后利用该字段类型在shcema.xml文件中增加一个新字段，如下例所示：

```
<fields>
  <!-- other fields -->
  <field name="wordPrefix" type="prefix" />
</fields>
<!-- other schema attributes -->
<copyField source="word" dest="wordPrefix"/>
```

现在，带有Word字段的文档在索引时其前缀存储在wordPrefix字段中。利用这些索引的字段，接下来就可以查询这些前缀。

从Solr中查找前缀匹配

利用该字段和字段类型，现在可以从Solr中对前缀进行查询并返回词表。这可以通过如下查询来实现：

```
http://localhost:8983/solr/select?q=wordPrefix:tam&fl=word
```

为从浏览器的搜索输入字段完成上述任务，当用户输入并展示高排名结果时，也需要撰写JavaScript代码来执行查询。之类的JavaScript代码涉及搜索字段中键盘敲击事件的处理，也涉及定期发送请求给服务器以返回扩展查询表并显示。幸运的是，这是一个再普通不过的行为，存在许多JavaScript库可供使用。其中一个比较流行的库是script.aculo.us（jQuery也支持这个功能）。它需要指定前缀查询并以非排序HTML列表形式返回扩展查询。尽管可以很容易编写一段小服务程序（servlet）来和Solr交互并格式化返回结果，我们可以把它当成展示Solr定制请求格式和响应输出能力的例子来使用。

定制响应格式需要编写一个QueryResponseWriter类，同时要指定它用于提前输入查询的响应实施中。编写一个定制的应答Writer十分直截了当，有多个实现都包含在Solr中。对于我们给出简单例子的代码如程序清单4-9所示。

清单4-9 对提前输入查询结果进行格式化的Solr响应写入器

```

public class TypeAheadResponseWriter implements QueryResponseWriter {
    private Set<String> fields;
    @Override
    public String getContentType (SolrQueryRequest req,
                                  SolrQueryResponse solrQueryResponse) {
        return "text/html;charset=UTF-8";
    }
    public void init (NamedList n) {
        fields = new HashSet<String> ();
        fields.add ("word");
    }
    @Override
    public void write (Writer w, SolrQueryRequest req,
                      SolrQueryResponse rsp) throws IOException {
        SolrIndexSearcher searcher = req.getSearcher ();
        NamedList nl = rsp.getValues ();
        int sz = nl.size ();
        for (int li = 0; li < sz; li++) {
            Object val = nl.getVal (li);
            if (val instanceof DocList) {
                DocList dl = (DocList) val;
                DocIterator iterator = dl.iterator ();
                w.append("<ul>n");
                while (iterator.hasNext ()) {
                    int id = iterator.nextDoc ();
                    Document doc = searcher.doc (id, fields);
                    String name = doc.get ("word");
                    w.append("<li>" + name + "</li>n");
                }
                w.append("</ul>n");
            }
        }
    }
}

```

指定应答 writer 展示的字段

寻找文档列表

以指定字段 返回文档

包含这个类的JAR文件必须放在Solr库的目录下，该目录通常为solr/lib，因此Solr可以解析该类。另外，也需要通知Solr有关你的应答Writer，并创建一个默认会使用该Writer的端点。上述做法可以在solrconfig.xml文件中实现，如程序清单4-10所示。

清单4-10 在Solr中指定应答Writer和请求Handler

```

<queryResponseWriter name="tah"
    class="com.tamingtext.fuzzy.TypeAheadResponseWriter"/>
<requestHandler name="/typeahead"
    class="solr.SearchHandler">
    <lst name="defaults">
        <str name="wt">taht</str>
        <str name="defType">dismax</str>
        <str name="qf"> wordPrefix^1.0 </str>
    </lst>
</requestHandler>

```

指定定制的
应答 Writer

以 URL 路径方式指定请求 Handler

指定默认的应答 Writer

指定搜索字段

上述新的请求Handler允许查询前缀并利用script.aculo.us进行合理的格式化处理。具体地，现在你可以使用下面的Solr查询，而不再需要指定应答类型或搜索字段作为查询参数。

清单4-11 使用查询应答Handler访问前缀的Solr URL

```
http://localhost:8983/solr/type-ahead?q=tam
```

动态填充搜索框

现在可以构建一个提前输入的搜索框，如程序清单4-12所示。

清单4-12 实现搜索提前输入功能的HTML和JavaScript

```

<html>
<head>
    <script src="./prototype.js" type="text/javascript">
    </script>
    <script src="./scriptaculous.js?load=effects, controls"
        type="text/javascript">
    </script>
    <link rel="stylesheet" type="text/css"
        href="autocomplete.css" />
</head>
<body>
    <input type="text" id="search"
        name="autocomplete_parameter"/>
    <div id="update" class="autocomplete"/>
    <script type="text/javascript">

```

导入 script.aculo.us

指定输入字段

为提前输入结果指定 div

```
new Ajax.Autocompleter ('search', 'update',           ← 构建提前输入对象
                        '/solr/type-ahead',
                        {paramName: "q", method:"get"});

</script>
</body>
</html>
```

JavaScript渲染的浏览器结果如图4-3所示。

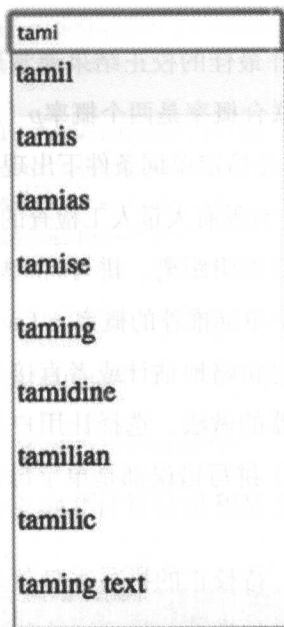


图4-3 对于前缀tami推荐的提前输入完整词

在上例中，可以看到如何使用前缀匹配在搜索应用中加入提前输入功能。你可以在Solr中加入一个字段和字段类型来存储前缀并使用Solr中的词条过滤器来产生特定大小的前缀。Solr的灵活性也能让你对请求和应答的处理进行定制，能够很容易地集成诸如script.aculo.us的JavaScript库。下一个例子中，我们会继续使用上一节介绍的模糊匹配工具来构建其他的应用。

4.3.2 搜索中的查询拼写校正

在网站上加入查询拼写校正功能有助于用户区分到底是查询拼错还是确实不能返回任何结果。该功能能够让用户修改查询的体验更加富有成效也更令人满意。

在本节中，我们会考察如何从概率角度对拼写及其校正结果排序，然后介绍如何利用前面介绍的工具和技术来实现对这些概率的近似估计。这很容易就可以通过使用Solr、SolrJ和Lucene提供的库来实现。尽管得到的拼写检查器可以在第一时间内用，至少它提供了一个基础，基于它可以对它及Lucene提供的拼写检查器进行改进。最后，我们会介绍如何利用Solr中提供的拼写校正模块，并且介绍该模块的架构如何会使集成定制的拼写模块十分容易。

方法概述

对于一个拼错的单词，选择最佳的校正结果通常形式化为一个求拼写和校正结果联合概率最大化的任务。该联合概率是两个概率 $p(s|w)$ 、 $p(w)$ 的乘积，其中 s 是拼写， w 是单词。前一个概率是给定单词条件下出现某种特定拼写的概率，而后一个概率是单词自己的概率。由于在没有大量人工检查的校正数据时估计 $p(s|w)$ 会很难，一种合理的估计方法是利用编辑距离。拼写 s 和单词 w 的归一化编辑距离可以看成是 $p(s|w)$ 的估计结果。某个单词推荐的概率 $p(w)$ 通常要比估计某个特定拼写的概率容易。很多情况下，通过粗略地估计或者直接忽略掉这个概率可以得到合理的结果。这是大部分拼写检查器的做法，选择让用户从列表中选出结果。这种做法的一种解释是大部分（80-95%）拼写错误都是单字母错误。编辑距离可以为推荐结果提供一种合理的排序方法。

上述做法失败的情况包括：待校正的拼写本身就是另一个词，或者合理推荐结果的数目太大，或者只能给出单条推荐结果。在这些情况下，基于其似然来影响排序结果会极大地提高性能。可以基于查询日志或者带词频的索引文本来实现这种思路。对于多词短语来说， n 元组模型可以用于估计单词序列，但是有关 n 元组模型的讨论不在本书范围之内。

既然现在已经对拼写校正的理论基础有所了解，下面我们考察如何实现。基于模糊匹配的拼写校正方法如下：

- 构建候选校正结果集合。
- 对这些候选结果评分。
- 使用阈值来显示推荐结果。

正如前面看到的那样， n 元组匹配是一种构建候选匹配的好方法。考察与待纠正拼写包含最多公共 n 元组匹配的项会产生一张很好的候选表。然后基于编辑距离将

这些候选词项重排序。最后，需要应用一个编辑距离阈值来保证，在没有合适推荐结果的情况下就不给出推荐结果。精确的阈值通常需要实验来设定。

在Solr中实现Did You Mean功能

可以利用Solr和SolrJ来实现前面的方法。首先需要定义一个字段类型和一个字段以及其在schema.xml文件中存储 n 元组的数据源，例子如程序清单4-13所示。

清单4-13 在Solr中为支持 n 元组匹配对Schema所做的修改

```
<fieldtype name="ngram" stored="false" indexed="true"
  class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.KeywordTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.NGramFilterFactory"
      minGramSize="2" maxGramSize="10"/>
  </analyzer>
</fieldtype>
<!-- other types -->
<field name="wordNGram" type="ngram" />
<!-- other fields -->
<copyField source="word" dest="wordNGram"/>
```

接下来需要对该字段进行查询并计算结果和某个具体拼写的编辑距离，具体代码如程序清单4-14所示。

清单4-14 从Solr中获取可能的校正结果并对它们进行排序的Java代码

```
public class SpellCorrector {
    private SolrServer solr;
    private SolrQuery query;
    private StringDistance sd;
    private float threshold;

    public SpellCorrector (StringDistance sd, float threshold)
        throws MalformedURLException {
        solr = new CommonsHttpSolrServer (
            new URL ("http://localhost:8983/solr"));
        query = new SolrQuery ();
        query.setFields ("word");
        query.setRows (50);
        this.sd = sd;
        this.threshold = threshold;
    }
}
```

需要考虑的 n
元组匹配次数

```

    }
    public String topSuggestion (String spelling)
        throws SolrServerException {
        query.setQuery ("wordNGram:"+spelling);
        QueryResponse response = solr.query (query);
        SolrDocumentList dl = response.getResults ();
        Iterator<SolrDocument> di = dl.iterator ();
        float maxDistance = 0;
        String suggestion = null;
        while (di.hasNext ()) {
            SolrDocument doc = di.next ();
            String word = (String) doc.getFieldValue ("word");
            float distance = sd.getDistance (word, spelling);
            if (distance > maxDistance) {
                maxDistance = distance;
                suggestion = word;
            }
        }
        if (maxDistance > threshold) {
            return suggestion;
        }
        return null;
    }
}

```

包含 n 元组的查询字段

计算编辑距离

保留最佳推荐结果

检查是否满足阈值，否则不返回推荐结果

SpellCorrector构造函数实现了org.apache.lucene.search.spell.StringDistance接口，当字符串相同时，传递给该函数的对象会返回1，而当两者最大限度不同时，返回0。在Lucene中有上述接口的多个实现，包括Levenshtein距离、Jaro-Winkler 距离和 n 元组距离。

刚才介绍的这个方法相当于忽略（或者看成常数）前面提到模型的单词概率。估计单词的概率可以通过计算单词在文档集中或查询日志中的数目并除以所有词的出现总数来实现。然后，这一项可以和编辑距离一起来用于排序。如果你希望该概率影响Solr返回的推荐结果，那么可以利用文档boosting来实现。

文档boost是一个提升文档相关度的乘法因子，它在索引时设立，通常大于1。该因子为2意味着该文档比没有因子情况下天生就相关2倍。由于在Solr中每条可能的推荐结果都被视作文档，并且由于单词的概率独立于拼写的概率，所以可以直接将 $p(w)$ 建模为文档的相关度。确定与你的领域相吻合的boost因子需要实验。当boost

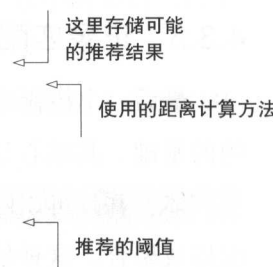
因子的值确定之后，它们就会影响前面使用的n元组查询的返回结果次序从而改变推荐结果。

利用Solr的拼写检查模块

现在已经理解了如何使用Solr实现拼写检查，接下来看看Solr内置的拼写校正机制。Lucene提供了一个拼写检查的实现并集成到Solr中。其方法和刚才介绍的方法类似。在Solr中可以作为搜索模块调用并添加到请求Handler中。拼写的搜索模块定义如程序清单4-15所示。

清单4-15 在Solr中将拼写检查器定义为搜索模块

```
<searchComponent name="spell_component"
  class="org.apache.solr.handler.component.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">word</str>
    <str name="distanceMeasure">
      org.apache.lucene.search.spell.LevenshteinDistance
    </str>
    <str name="spellcheckIndexDir">./spell</str>
    <str name="accuracy">0.5</str>
  </lst>
</searchComponent>
```



其可作为一个参数在Handler默认定义后加入到请求Handler中。

清单4-16 在Solr将拼写校正搜索模块加入到请求Handler中

```
<requestHandler ...
  <lst name="defaults">
    ...
  </lst>
  <arr name="last-components">
    <str>spell_component</str>
  </arr>
</requestHandler>
```

提交给拼写检查器的查询会同常规查询及常规搜索结果的推荐结果一起处理。这对于允许一条推荐结果及单条给Solr的请求返回一条结果十分有益。在拼写校正需要不同与请求Handler（如dismax Handler）的切词时，拼写查询可以置于另一参数spellcheck.q。

正如前面提到的那样，Solr中提供、Lucene中发现的拼写校正模块使用的方法与我们基于SolrJ实现的方法类似。有些不同在于为校正结果的前缀匹配加入的额外的权重，并加入了只推荐那些频率高于查询词项的词的功能。你可以构建一个定制的拼写检查模块并集成到你的Solr编译当中去。这可以通过继承抽象类`org.apache.solr.spelling.SolrSpellChecker`并实现`getSuggestions`方法及构建和重载模块的方法来实现。如前所述，该类必须包含在Solr的`solr/lib`目录下的JAR中以便可供Solr使用。这样之后，该模块可以像`SpellCheckComponent`类那样在配置中指定。

本节我们讨论了如何将基于 n 元组的模糊匹配技术和编辑距离的计算技术结合在一起来进行拼写校正。接下来我们将利用这些技术以及其他一些技术在一个记录匹配任务中对更大范围的字段进行模糊匹配。

4.3.3 记录匹配

最后一个模糊串匹配的应用并不像前面的例子那样突出，但是却是很多有趣应用的基础。其核心是基于应用混搭的记录匹配。如果有两个数据源包含实际中的同一实体，并且可以匹配这些记录，那么就可以将每个数据源中包含的不同信息进行混搭或聚合。这种信息的组合往往可以提供一个截然不同的视角，即单独信息源的任何一方都不能提供全部的信息。某些情况下，这个结论相当直观，但是在很多情况下需要做一些模糊匹配。

方法概述

基于模糊匹配的记录匹配方法如下。

- 寻找候选匹配。
- 对候选匹配排序或评分。
- 对结果进行评估并选择一个候选结果。

这与前面的拼写校正方法类似，那里我们通过 n 元组匹配来确定候选匹配，利用编辑距离来对候选匹配评分，最后通过阈值来选出候选匹配。这里主要增加了一条准则，即只寻找单条超过阈值的匹配。在有多条候选结果且算法并不能对它们清晰排序时，这种做法能够防止匹配被断言。

我们的例子来自影院上映的影片。这个领域中有许多不同信息源都包含了对相同空间的引用。这里包括的数据源有Internet Movie Database（IMDb）、NetFlix、

TV以及点播表、iTunes和Amazon Rental或DVD。在这个例子中，我们将对IMDb和Tribune Media Service（TMS）的影片进行匹配。TMS为Tivo提供了TV清单数据，在线地址为<http://tvlistings.zap2it.com>。

利用Solr寻找候选匹配

首选需要找到一种方法来识别匹配候选集，这样就可以在上面进行更高级的匹配。像拼写校正一样，你可以使用Solr中的n元组匹配来实现。这里将n元组匹配应用到你认为最有可能匹配并且有最大信息量的字段。对于影片来说，该字段为片名。和拼写校正中可以明确选择哪个数据集应用n元组（校正结果）不同的是，这里需要对两个数据源都构建n元组词条。尽管基于具体应用的原因有些数据源可能比另一个要优先，通常选择具有最多条目数的数据源应用n元组更有利。这是因为会在条目上运行多次匹配算法而n元组构建改变的可能性较小。提高算法时，在小的条目集合上循环会更快。此外，索引往往比记录匹配快。

对于你的数据集而言，在我们的IMDb数据库中对每部影片已经构建了一个XML文档。一个典型的条目如下所示。

清单4-17 我们IMDb数据库中的条目示例

```
<doc>
  <field name="id">34369</field>
  <field name="imdb">tt0083658</field>
  <field name="title">Blade Runner</field>
  <field name="year">1982</field>
  <field name="cast">Harrison Ford</field>
  <field name="cast">Sean Young</field>
  <!-- Many other actors -->
</doc>
```

这些字段在schema.xml条目中的相关部分如下所示。

清单4-18 针对记录匹配在Solr schema中的添加项

```
<field name="title" type="ngram"
  indexed="true" stored="true"/>
<field name="year" type="integer"
  indexed="true" stored="true"/>
<field name="imdb" type="string"
```

前面拼写校正中使用的n元组字段

```
indexed="false" stored="true"/>
<field name="cast" type="string" indexed="true"
multiValued="true" stored="true"/>
```

← 多值 cast (指演员) 字段

利用SolrJ可以查询并返回候选结果，代码如下所示。

清单4-19 从Solr返回记录匹配的候选匹配结果

```
private SolrServer solr;
private SolrQuery query;
public Iterator<SolrDocument> getCandidates (String title)
    throws SolrServerException {
    String etitle = escape (title) ;
    query.setQuery ("title:"+"etitle+"") ;
    QueryResponse response = solr.query (query) ;
    SolrDocumentList dl = response.getResults () ;
    return dl.iterator () ;
}
```

← 转义后的片名

← 片名放在引号中避免被切词

片名必须要转义以避免一些特殊字符在当做查询使用时（比如AND、+、!）被解释成查询函数。

对候选匹配排序

在拥有候选匹配集合之后，就看编程如何对这些匹配评分。在拼写校正中，使用的是编辑距离。这对于片名来说也是个很好的候选做法，但是通过利用不同字段的数据可以达到最优匹配的结果。考虑如下想要匹配的字段，并考虑如何对每个元素评分和整条记录的匹配评分。

- 片名：这里最合适的计算方法可能就是编辑距离。由于片名更像姓名，可以使用Jaro-Winkler距离而不是Levenshtein距离或 n 元组距离。
- 演员：尽管可以使用编辑距离来计算演员的姓名，由于演员的姓名实际就是他们的实际品牌，因此倾向于使用标准的拼写（别人不可能是演员Thomas Cruise）。因此，带点归一化处理的精确匹配可能会在大部分情况下奏效。演员重合度是精确匹配的演员比例。由于不同数据源之间的演员数目往往会有差别，通常可以接受利用其中较小的演员数目作为分母。
- 发布日期：发布日期对于区分具有相同片名的影片也十分有用。这些日期之所以有微小差别取决于该日期到底是项目开始、还是影院上映或者DVD发布

的年份。不同的年份之间的差异可以通过倒数排名的方法来进行归一化，同一年份的会得1分，年份相差2年得1/2。如果可以建立评分为0或其他常数的临界点，这种年份的差异也可以按照线性衰减的方式来计算。

之后需要对上述所有的得分进行合并，这可以通过对不同得分加权求和来完成。由于每一部分的得分都归一化处理到0到1之间，因此如果选择的权重之和也是1的话，那么加权求和的结果也将在0到1之间。对于手边的这个任务，可预先分配一半权重给片名，然后将剩余的权重分配给其他两个部分。上述过程在清单4-20中实现。你也可以利用已匹配记录构成的文档集来经验性确定这些方法。通过记录的片名查询Solr对这些候选影片进行排名，从而实现最终的匹配过程。

清单4-20 从Solr中返回记录匹配任务的候选匹配

```
private StringDistance sd = new JaroWinklerDistance ();
private float score (String title1, int year1, Set<String> cast1,
    String title2, int year2, Set<String> cast2) {
    float titleScore = sd.getDistance (title1.toLowerCase (),
        title2.toLowerCase () );

    float yearScore = (float) 1/ (Math.abs (year1-year2) +1 );
    float castScore = (float) intersectionSize (cast1, cast2) /
        Math.min (cast1.size (), cast2.size () );

    return (titleScore*.5f) +
        (yearScore*0.2f) +
        (castScore*0.3f );
}

private int intersectionSize (Set<String> cast1,
    Set<String> cast2) {
    int size = 0;
    for (String actor : cast1)
        if (cast2.contains (actor)) size++;
    return size;
}
```

片名使用 Jaro-Winkler 距离

用年份差的倒数

使用演员的重合率

将多个得分合并成一个得分

通过精确字符串匹配计算交集

结果评估

下面看看利用上述方法的一些结果示例。一条可以反映多数据集合并匹配优势的例子如表4-1和4-5所示。

表 4-4 体现多个数据集合并重要性的例子

ID	片 名	年 份	演 员
MV000000170000	Nighthawks	1981	Sylvester Stallone, Billy Dee Williams, ...

表 4-5 体现多个数据集合并重要性的第二个例子

得 分	片名得分 + 年份得分 + 演员得分	ID	片 名	年 份	演 员
0.55	$(0.5 * 1.00) + (0.2 * 0.25) + (0.3 * 0.00)$	tt0077993	Nighthawks	1978	Ken Robertson, TonyWestrope, ...
0.24	$(0.5 * 0.43) + (0.2 * 0.12) + (0.3 * 0.00)$	tt0097487	Hawks	1988	Timothy Dalton, AnthonyEdwards, ...
0.96	$(0.5 * 0.98) + (0.2 * 1.00) + (0.3 * 0.88)$	tt0082817	Night Hawks	1981	Sylvester Stallone, Billy DeeWilliams, ...

Solr只基于片名匹配返回上述候选影片。你会看到，数据的组合不仅可以正确结果排在其他结果之前，而且能够排除那些候选结果，这样你就对最高得分的匹配拥有很大把握。通常来说，利用 n 元组方法来返回匹配结果以及利用Jaro-Winkler编辑距离允许处理数据中的一些变异，比如标点符号、数字、副标题甚至错误拼写等。下面列出了这样的一些影片的片名：

- *Willy Wonka and the Chocolate Factory and Willy Wonka & the Chocolate Factory*
- *Return of the Secaucus 7 and Return of the Secaucus Seven*
- *Godspell and Godspell: A Musical Based on the Gospel According to St. Matthew*
- *Desert Trail and The Desert Trail*

将该方法用于TMS的1000部影片，可以匹配到IMDb中的884部影片。这些全都是正确的匹配，因此在假设所有影片都能找到匹配的条件下，此时的正确率为100%，召回率为88.4%。这也意味着由于利用现有算法和权重没有碰到任何错误，因此如果你调整算法，就可以实现比较宽松的匹配。尽管上述例子的目标并不是优化影片匹配，了解一下上述算法没有找到匹配的原因有助于在构建其他领域的记录匹配

算法时考虑另外一些因素。表4-6列出了一些失配的例子。

表 4-6 失配的例子

TMS 片名 / 年份	IMDb 片名 / 年份	描 述
M*A*S*H (1970)	MASH (1970)	这种情况下, 由于其中一个片名的全部 n 元组都包含星号, 而另一个片名的 n 元组则不包含星号, 因此基于 n 元组的匹配会失败
9 to 5 (1980)	Nine to Five (1980)	这种情况下, 数字必须要进行归一化处理才能匹配两个片名
The Day the World Ended (1956)	Day the World Ended (1955)	这种情况下, 由于其他字段不匹配 (比如日期), 所以首部限定词不匹配的错误无法得到补偿。将片名首部的 The、An、A 去掉会缓解该问题
Quest for Fire (1981)	<i>La guerre du feu</i> (1981)	某些情况下不论做多少归一化都无济于事。本例要么通过人工编辑来解决, 要么换个片名
<i>Smokey and the Bandit</i> (1977)	<i>Smokey and the Bandit</i> (1977)	这种情况下, 尽管最佳匹配就是正确匹配, 但是该影片和其续集 <i>Smokey and the Bandit II</i> 的得分也在阈值之上, 因此本匹配被取消
<i>The Voyage of the Yes</i> (1972)	<i>The Voyage of the Yes</i> (1972)	本例中, 一个数据源 (TMS) 将此归于电影类, 而在另一个数据源 (IMDb) 中将此归于 TV 秀。因此, 我们并非候选匹配

上述情况展示了即使使用最好的技术, 数据归一化处理对于成功匹配都至关重要。算法会从多个归一化过程中受益, 包括数字、限定词和替代的片名的归一化等等。为产生高质量的算法, 需要花费相当的精力来确保, 进入你的匹配代码的是你希望的数据。

本节当中, 我们介绍了将多种字符串匹配技术用于一系列应用使它们更加强大的方法。为支持提前输入, 我们使用Solr中的前缀匹配。我们展示了如何组合 n 元组匹配和编辑距离来推荐可能的拼写结果。最后, 我们使用 n 元组匹配、编辑距离和精确匹配来对影片进行记录匹配。

4.4 小结

本章一开始就提出“字符串相似是什么意思”这个问题, 即模糊匹配有多模

糊？然后介绍了多种模糊字符串匹配的方法，并通过它们给出了两个字符串相似的正式概念。这些方法包括只使用字符的方法（比如Jaccard距离）、基于字符顺序的方法（比如编辑距离）、基于字符窗口的方法（比如Jaro-Winkler距离和 n 元组编辑距离）。我们也展示了前缀匹配和 n 元组匹配可以在计算开销更大的编辑距离计算时高效产生候选匹配。最后，我们利用这些技术构建了应用，并利用Solr作为平台来使得这些应用的构建十分容易。下一章将从字符串对比转到从字符串和文档内部来寻找信息。

4.5 相关资源

Aoe, Jun-ichi. 1989. “An efficient digital search algorithm by using a double-arraystructure.” IEEE Transactions on Software Engineering, 15, no.9:1066–1077.

第5章 命名实体识别

本章内容

- 命名实体识别的基本概念
- 如何使用OpenNLP来识别命名实体
- OpenNLP的性能问题

人、地点、事物或者说名词在语言中担任重要角色，它们承载句子的主语，并且往往也承载句子的宾语。出于其重要性，在文本处理时识别并在应用中使用名词往往十分有用。该任务往往称为实体识别或命名实体识别（简称NER），常常通过句法分析器或组块分析器来实现，第2章已经对此有所介绍。尽管利用分析器十分有助于句子的理解，在文本应用中却往往发现其中的某个名词子集更有用，这类名词给出了对象（如专有名词）的具体实例，通常也叫做命名实体。进一步而言，完整句法分析是一个过程密集型任务，而寻找专有名词则并不需要达到这种密集程度。

很多情况下，除人名、地名之外还有很多实体也很有用，比如时间（July 2007）或者数字（\$50.35）。从文本应用的角度来看，专有名词无处不在，但与此同时，某个专有名词的具体实例却又可能极其罕见。比如，考虑任意近期发生的新闻事件（特别是那些不涉及明星或高官的事件）。计算一下，在该事件的新闻报道中会有多少专有名词？其中有多少人以前没听说过？六个月后这些人当中还有多少仍然会在新闻中出现？这些事件发生在什么时间和地点？

很明显，文章的上下文信息会提示某个特定的词序列是一个专有名词，并且也有可能存在一些其他的线索，比如字母大写或者像Mr.或Mrs.一样的称谓，但是如何找到其中的规律以便能够在文本处理应用中学到实体识别的方法？本章首先会花些时间来理解命名实体识别的背景，然后考察能够学习命名实体识别的Apache OpenNLP工具。最后也考察该工具的性能问题以及为某个领域定制模型。下面首先看看命名实体的作用。

人名、机构名、地名和其他命名实体的识别能够使得捕捉文章的主题变得切实可行。例如，利用该信息可以为这些实体提供更多信息，推荐其他也以这些实体为重点或与它们相关的内容，最终能够增加网站的交互度¹。在很多大公司或机构，命名实体识别的工作往往通过人工编辑来完成。结果可能会是这样一个网站：人们沉浸于该网站文章某个部分的阅读中，看到另一个有趣的链接就会漫游到另一篇文章当中，只有看到时钟，不然他们不会意识到近一个小时已经花在该网站上。例如，图5-1中，Yahoo!对命名实体*Sarah Palin*进行了高亮显示，并增加了一个弹出窗口用于强调与这位2008年副总统候选人有关的其他内容。甚至在弹窗的底部还基于该命名实体展示了相关广告以期获得回报。这种交互度对于网站（特别是那种通过广告展示来套现的网站）来说是无价之宝，培育该交互度能够提高用户回访网站的可能。可以想象的是，人工编辑的方法需要消耗大量人力，公司往往寻求自动、或者至少半自动的方法来实现命名实体的识别。

进一步来说，与关键词、标签或者其他基于意义的文章内容表示不同，基于实体出现与否的文章关联概念所体现的关系（假定像第4章一样进行了正确的记录匹配来确保两个实体相同）更清晰，对用户来说也很直观。

本章会考察如何在文本中自动识别命名实体。我们会考察一个流行的开源命名实体识别工具的精度及运行性能问题，以便有助于确定应用该工具的合适时间和地点。我们也会考察如何对其模型进行定制以便在自己的数据上更加有效。即使不想将命名实体信息直接暴露给用户，这些信息对于构建诸如网站关键词排名榜或者网站提到的十大流行人名排行榜这样的数据都十分有用。了解命名实体这些可能的用

1 “交互度”衡量的是访问者和网站的交互（点击、上传/播放、下载、留言、参与交互游戏等）程度，该术语参考了网站<http://www.chinawebanalytics.cn/>的翻译。有兴趣者可以访问该网站内容。——译者注

途之后，下面对其细节进行深入阐述。

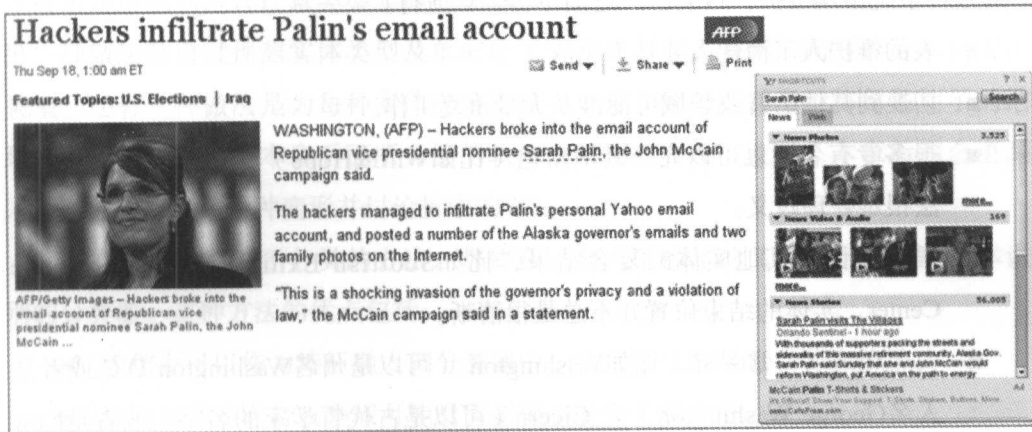


图5-1 Yahoo!新闻的片段及实体页面链接的例子。Sarah Palin标记为Yahoo!新闻的一个快捷方式。该图截自2008年9月21日

5.1 命名实体的识别方法

在命名实体识别 (named-entity recognition, 简称NER) 中, 我们感兴趣的是某些或全部提及识别人名、地名、机构名、时间和数字 (严格来说, 这些实体不一定是专有名词, 但为了简单起见, 这里都把它们归为专有名词)。NER的结果可以用于回答有关where、when、who、how often或how much的问题。例如, 给定句子The Minnesota Twins won the 1991 World Series, 某个NER系统可能会把Minnesota Twins、1991和World Series一块识别为命名实体 (或者有可能把1991 World Series识别为单个命名实体)。实际上, 并非所有的系统对命名实体抽取的需求都一模一样。比如, 营销者可能会查找他们产品的名称以便能够了解哪些人在谈论这些产品, 而一位基于数百名目击者的叙述来负责编撰事件表的历史学家不仅对事件中的人物感兴趣, 而且对每个人目睹事件的精确时间和地点也感兴趣。

5.1.1 基于规则的实体识别

一种进行NER处理的办法是组合使用表和正则表达式来识别命名实体。利用这种方法, 只需要编写一些基本的有关大写和数字的规则, 然后与某些表 (这些表包含诸如常见名、常见姓、热门地点、星期、月份等信息) 混合使用, 最后输出一串

串的文本。这种方法在早期的命名实体识别系统研究中十分流行，但是基于如下的原因这种系统很难维护，目前已经逐渐不再那么流行。

- 表的维护人工消耗大并且不灵活。
- 切换到其他语言或领域可能涉及大量重复工作。
- 很多专有名词也可以充当其他角色（比如Will或Hope）。换句话说，上述方法很难处理歧义。
- 很多实体是其他实体的复合结果，比如Scottish Exhibition和Conference Center，实体的结束位置并不总是很清晰。
- 人名和地名常常一样，比如Washington（可以是州名Washington D.C.或者是人名George Washington）或Cicero（可以是古代哲学家的名字，或者是New York的一个镇名或者其他地方的地名）。
- 利用正则表达式规则方式很难对文档内实体之间的依赖关系建模。

需要指出的是，基于规则的方法在某些已充分了解的特定领域效果不错，因此并不能被完全抛弃。比如在长度测量领域，东西本身通常很少而长度的单位也很有限，因此这种方法可能适用。有很多有用的公开资源有助于引导对一系列实体类型进行上述处理。实体识别有关的基本规则和一些通用资源可以参考美国中央情报局的《世界概况》（CIA World Fact Book, <https://www.cia.gov/library/publications/the-world-factbook/index.html>）和维基百科（<http://www.wikipedia.org>）。网上也有多部专有名词词典，同时也有很多领域相关的资源，如互联网影片数据库（Internet MovieDatabase，简称IMDb）或者一些领域知识库，它们可以高效利用以获得较好性能，同时将所需的工作量降到最小。

5.1.2 基于统计分类器的实体识别

另一种实体识别方法不那么脆弱，它能够很容易扩展到其他领域或语言，并且不需要构建并维护大表（如地名词典）。这种更为可取的方法利用统计分类器来识别命名实体。通常地，分类器会考察句子中的每个词，确定该词处于命名实体的首部、已开始命名实体的中部还是根本不属于当前命名实体。通过合并这些预测结果，就可以使用分类器来识别构成实体的词序列。

尽管上述标注方法对于大部分基于分类器的实体识别方法而言相当普遍，不同

的实体类型的识别还是存在差异。一种做法是第一遍使用标注方法甚至正则表达式方法来简单识别包含任意类型实体的文本，然后在第二遍中区分不同的实体类型。另一种做法是通过预测实体类型及单词处于实体首部还是中部来同时区分不同实体类型。还有一种做法是为每种实体类型构建一个独立的分类器然后将每条句子的结果进行合并。这也是本章后面所介绍的软件中采用的方法。此外，本章后面会更深入考察上述方法的各种变形并讨论折中的方法。

不论使用哪种基于分类的方法，都要在一个人工标注的文本集上训练出实体识别的分类器。这种方法的优点包括：

- 表信息可以作为特征融入到分类器中，它只是分类器的一类信息源。
- 切换到其他语言或领域可能只需要很小的代码修改量。
- 对句子内和文档内的上下文建模更加容易。
- 可以通过重新训练分类器来融入额外的文本或其他特征。

这种方法的主要缺点是需要人工标注的数据。程序员写一些规则然后很快就能看到它们在文本集上的应用效果，而与此不同的是，分类器往往需要大约3万词左右的语料上进行训练才能获得较好的结果。尽管标注过程冗长乏味，但是并不需要编写规则所需要的专家，并且标注后的资源可以扩展和重用。利用足够的训练数据，NER的性能能够接近人的水平，当然人类在实体识别上也没有那么完美。好的NER系统通常可以在评估实验中取得超过90%以上的正确率。实际在真实数据上的预期肯定要低一点，但是大部分系统仍然可以取得相当好的可用结果。进一步而言，搭建一个好的系统比较容易，需要的话，可以训练成专有名词的识别工具。理想地，当有新样本或反例时，系统也支持增量式更新。考虑到这些需要，下一节将考察OpenNLP项目如何提供命名实体识别功能。

5.2 基于OpenNLP的基本实体识别

第2章提到的OpenNLP项目目前可以从地址<http://opennlp.apache.org>下载，它维护了一套处理常见NLP任务的工具，这些任务包括词性标注、句法分析以及本章中最有用的命名实体识别等等。这些工具的使用许可证是Apache Software License (ASL)，最早由Thomas Morton等人开发，但是现在和Solr一样都属于Apache软件基金会的一个项目，被用户和贡献者社区所维护。尽管存在多个工具可以完成命名

实体识别任务，但是它们中的大部分要么不开源，要么属于研究项目，很多工具只有仅供研究使用的许可证或者GPL，这样的话很多公司往往并不认为其具备可用性。OpenNLP的发布版本中带有多个模型，这些模型对于常见的实体类型效果很好，并且OpenNLP也被积极地维护和支持。基于上述原因，并且也基于本书作者对软件的熟悉度，下面将集中关注OpenNLP所提供的命名实体识别功能。

OpenNLP发布时自带了内建模型，允许识别专有名词和数量并按照语义将它们分到七种不同类别。这些类别及文本样例如下所示。

- 人名：Bill Clinton, Mr. Clinton, President Clinton
- 地名：Alabama, Montgomery, Guam
- 机构名：Microsoft Corp., Internal Revenue Service, IRS, Congress
- 日期：Sept. 3, Saturday, Easter
- 时间：6 minutes 20 seconds, 4:04 a.m., afternoon
- 百分比：10 percent, 45.5 percent, 37.5%
- 货币：\$90, 000, \$35 billion, one euro, 36 pesos

用户可以根据具体项目的要求选择上述类别集合的子集。

本章其余部分将介绍如何利用OpenNLP从文本中识别前面提到的实体类型，然后将考察一些有助于理解抽取结果的工具。最后，考察如何利用OpenNLP给出的分数来了解某个或多个抽取结果正确的可能性。

5.2.1 利用OpenNLP寻找人名

首先介绍一个例子，看看如何通过编写一小段Java代码利用OpenNLP来识别人名。

清单5-1 利用OpenNLP识别人名

```
String[] sentences = {  
    "Former first lady Nancy Reagan was taken to a " +  
        "suburban Los Angeles " +  
    "hospital "as a precaution" Sunday after a " +  
        "fall at her home, an " +  
    "aide said. ",  
    "The 86-year-old Reagan will remain overnight for " +  
    "observation at a hospital in Santa Monica, California, " +
```

```

        "said Joanne " +
        "Drake, chief of staff for the Reagan Foundation.");
    NameFinderME finder = new NameFinderME (
        new TokenNameFinderModel (new FileInputStream (getPersonModel ()))
    );

    Tokenizer tokenizer = SimpleTokenizer.INSTANCE;
    for (int si = 0; si < sentences.length; si++) {
        String[] tokens = tokenizer.tokenize (sentences[si]);
        Span[] names = finder.find (tokens);
        displayNames (names, tokens);
    }
    finder.clearAdaptiveData ();

```

基于文件 en-ner-person. bin 的二进制压缩模型来初始化人名识别的新模型

切词器，该将句子分
和标识符

将句子切分
成词条数组

清除一些数据结构，这些
数据结构保存了前面的
单词以及这些词是否为
人名的一部分这种信息

识别句子中
的人名并返
回这些人
的词条偏移

本例中，首先创建一篇由两个句子组成的文档，然后对应用于该文档的 NameFinderME 类及切词器进行初始化。NameFinderME 给定了一个识别特定命名实体类型（这里是人名）的模型，该模型基于 OpenNLP 自带的人物模型文件。之后，对每个句子进行切词，其中的人名被识别并展示给用户。最后，当文档中的所有句子处理之后，要调用 clearAdaptiveData () 方法，它通知 NameFinderME 清除所有迄今为止在处理中存储的文档级数据。默认情况下，OpenNLP 的 NameFinderME 类会追踪某个单词是否先前被识别为实体的一部分，该信息对于判断后续内容是否仍然是实体的一部分来说十分重要。调用 clearAdaptiveData () 会清除上述缓存信息。

上例表明 NameFinderME 一次只处理一个句子。尽管并非明确要求，这样做能够避免识别出跨句边界的错误实体。通常而言，让实体识别工具处理最小的文本单位非常有益，这样能够避免将某个实体割裂开来。这是因为 OpenNLP 的实现实际上对处理的每个文本单位最多考虑三种实体类别。如果以文档为处理单位，那么对于整篇文档只有三种选择可能，但是如果处理句子，那么对每个句子就有三种可能。

NameFinderME.find () 方法的输入是一个词条序列。这也意味着对每个待处理的句子也必须要做切词处理。这里使用的 OpenNLP 提供的切词器能够基于字符类别来分割词条。由于切词会影响 find () 方法的句子输入，因此在新实体识别时要使用模型训练时的相同切词过程，这一点相当重要。在 5.5 节中，我们会讨论使用另外的切词器训练新命名实体识别模型的方法。

5.2.2 OpenNLP识别的实体解读

NameFinderME.find () 方法会返回一系列 Span 类型数据构成的数组，每个 span 给出了输入句子中识别的实体位置。OpenNLP Span 数据类型存储了实体第一个词条的下标（可以通过getStart ()方法获得），以及实体最后一个词条后面那个词条的下标（可以通过getEnd ()方法获得）。这里 Span 用于表示词条的偏移，但是OpenNLP也使用该数据类型来表示字符的偏移。下面的代码每行依次打印一个词条序列构成的实体。

清单5-2 利用OpenNLP显示实体

```
private void displayNames (Span[] names, String[] tokens) {
    for (int si = 0; si < names.length; si++) {
        StringBuilder cb = new StringBuilder ();
        for (int ti = names[si].getStart ();
            ti < names[si].getEnd (); ti++) {
            cb.append (tokens[ti]).append (" ");
        }
        System.out.println (cb.substring (0, cb.length () - 1));
        System.out.println ("ttype: " + names[si].getType ());
    }
}
```

在每个实体上循环

在实体中的每个词条上循环

去掉实体尾部的额外空间并打印

OpenNLP也提供了一项功能，即通过Span.spansToStrings ()方法将 Span 类型转换为表示实体的字符串，如清单5-3所示。

如果想看看实体未切词前的形式，可以通过如下所示的代码将实体映射到其字符偏移。这种情况下，利用tokenizePos ()方法可以要求切词器返回词条的字符偏移而不是词条的字符串表示。这可以确定实体在原始句子中出现的位置。

清单5-3 利用区间类显示实体

```
for (int si = 0; si < sentences.length; si++) {
    Span[] tokenSpans = tokenizer.tokenizePos (sentences[si]);
    String[] tokens = Span.spansToStrings (tokenSpans, sentences[si]);
    Span[] names = finder.find (tokens);

    for (int ni = 0; ni < names.length; ni++) {
        Span startSpan = tokenSpans[names[ni].getStart ()];
        int nameStart = startSpan.getStart ();
    }
}
```

切分成词条，返回字符偏移（区间）

在每个句子上循环

将区间转换为字符串

计算实体的首字符下标

识别实体，返回基于词条的偏移


```

    Span endSpan = tokenSpans[names[ni].getEnd () - 1];
    int nameEnd = endSpan.getEnd ();

    String name = sentences[si].substring (nameStart, nameEnd);
    System.out.println (name);
}
}

```

计算实体的尾字下标 (最后一字符的下标 +1)

计算表示实体的字符串

5.2.3 基于概率过滤实体

OpenNLP使用了概率模型来确定具体实体的识别概率。这对于某些情况下返回结果中过滤掉一些实体特别重要, 这些过滤掉的实体可能是错误的识别结果。尽管没有办法能够自动确定哪些实体被错误识别, 通常而言, 被模型赋予较低概率的实体被正确识别的可能性较低。为确定某个具体实体的概率, 在每个句子处理之后可以调用NameFinderME.getProbs () 方法, 如清单5-4所示。返回的值数组与以 Span 为输入识别出的实体数组的下标相对应。

清单5-4 确定实体的概率

```

for (int si = 0; si < sentences.length; si++) {
    String[] tokens = tokenizer.tokenize (sentences[si]);
    Span[] names = finder.find (tokens);
    double[] spanProbs = finder.probs (names);
}

```

句子分割
词条数组

对每个句子循环

识别实体, 返回基于词条的偏移

返回每个实体对应的概率

然后, 基于应用的需求确定一个概率阈值, 之后将低于阈值的实体去掉。

本节介绍了如何利用OpenNLP来识别单个的实体类型, 解释了OpenNLP中所使用的给出实体位置的数据结构, 最后确定了更可能正确的实体。下一节将会考虑多个实体的识别, 并深入探讨OpenNLP如何实际确定文本中实体的存在与否的细节。

5.3 利用OpenNLP进行深度命名实体识别

迄今为止我们已经了解了命名实体识别的基本知识, 接下来将考察一些更高级的例子, 当利用这些工具构建实际系统时更有可能遇到这些例子。正如5.1节所述, 有多种识别命名实体的方法。现有例子的一个主要局限性在于, 它们只涉及单类命名实体。本节会介绍如何利用OpenNLP在同一个句子中识别多种命名实体类型, 同

时介绍被每个类型所利用的信息。

5.3.1 利用OpenNLP识别多种实体类型

在OpenNLP中，每种实体类型使用自己独立的模型来进行识别。这样做的优点在于只要使用具体应用所需的模型子集，并且可以为其他实体类型添加自己的模型。这也意味着不同的模型有可能从文本的重合部分中识别出命名实体，如下所示：

```
<person> Michael Vick </person>, the former <organization><location>
Atlanta </location> Falcons </organization> quarterback, is serving a 23-
month sentence at maximum-security prison in <location> Leavenworth </
location>, <location> Kansas </location>.
```

这里会看到，*Atlanta*在标记为地名的同时也标记为机构名的一部分。

这种做法的缺点在于需要组合每个模型的结果。本节将会考察解决该问题的一些办法。每个实体类型都使用自己的模型也会影响性能和训练过程。在5.4节和5.5节中将会讨论有关这些影响的更多内容。

由于每个模型和其他模型相互独立，在每个模型的情况下使用多模型是一件再简单不过的事，而结果的组合则没有那么简单。在清单5-5中，我们收集了三个模型的实体结果。为促进这种做法，我们构建了一个辅助类Annotation来保存实体Span及其概率和类型。

清单5-5 在同一文本上运行多个命名实体模型

```
String[] sentences = {
    "Former first lady Nancy Reagan was taken to a " +
        "suburban Los Angeles " +
    "hospital "as a precaution" Sunday after a fall at " +
        "her home, an " +
    "aide said. ",
    "The 86-year-old Reagan will remain overnight for " +
    "observation at a hospital in Santa Monica, California, " +
        "said Joanne " +
    "Drake, chief of staff for the Reagan Foundation."};
NameFinderME[] finders = new NameFinderME[3];
String[] names = {"person", "location", "date"};
for (int mi = 0; mi < names.length; mi++) {
```

基于文件
en-nerperson.bin、
en-nerlocation.bin
以及 en-ner-date.
bin 文件中的二进制
压缩模型来对新的人
名、地名和日期识
别模型进行初始化

```

finders[mi] = new NameFinderME (new TokenNameFinderModel (
    new FileInputStream (
        new File (modelDir, "en-ner-" + names[mi] + ".bin")
    )));
}

Tokenizertokenizer = SimpleTokenizer.INSTANCE;
for (intsi = 0; si<sentences.length; si++) {
    List<Annotation>allAnnotations = new ArrayList<Annotation> ();
    String[] tokens = tokenizer.tokenize (sentences[si]);
    for (int fi = 0; fi <finders.length; fi++) {
        Span[] spans = finders[fi].find (tokens);
        double[] probs = finders[fi].probs (spans);
        for (intni = 0; ni<spans.length; ni++) {
            allAnnotations.add (
                new Annotation (names[fi], spans[ni], probs[ni])
            );
        }
    }
    removeConflicts (allAnnotations);
}

```

获得一个切词器的引用, 该切词器可以将句子分割成独立的单词和符号

将句子分割成词条数组

在每个句子上循环

识别句子中的命名实体并返回基于词条的偏移位置

从命名实体识别器中收集识别的实体

对重叠实体进行解析以得到更多可能的实体

每个命名实体人名、地名、日期识别器上循环

获得相关匹配及其概率

合并上述三个模型的输出问题时, 只有命名实体重叠。根据应用的不同, 重叠发生的判断准则也不相同。为合并上述结果, 需要考虑如下问题:

- 同一文本Span被不同模型都判断为命名实体是否可以? 通常来说, 不行。
- 小的命名实体包含在大的命名实体之中是否可以? 通常来说, 可以。
- 重叠的命名实体包含截然不同的文本是否可以? 通常来说, 不行。
- 如果实体冲突, 那么采用何种准则来确定最终结果呢? 通常来说, 是概率。

下面的清单给出了如下默认准则的实现: 不同实体必须属于不同的 Span, 可以重叠, 但是一旦重叠, 一个实体必须完全包含另一个实体。

清单5-6 冲突实体的解析

```

private void removeConflicts (List<Annotation>allAnnotations) {
    java.util.Collections.sort (allAnnotations);
    List<Annotation> stack = new ArrayList<Annotation> ();
    stack.add (allAnnotations.get (0));
    for (intai = 1; ai<allAnnotations.size (); ai++) {
        Annotation curr = (Annotation) allAnnotations.get (ai);
    }
}

```

初始化栈以跟踪先前的命名实体

在每个命名实体上循环

先基于 Span 开始下标的升序后基于结束下标的降序对命名实体排序

```

boolean deleteCurr = false;
for (int ki = stack.size () - 1; ki >= 0; ki--) {
    Annotation prev = (Annotation) stack.get (ki) ;
    if (prev.getSpan () .equals (curr.getSpan ())) {
        if (prev.getProb () > curr.getProb ()) {
            deleteCurr = true;
            break;
        } else {
            allAnnotations.remove (stack.remove (ki)) ;
            ai--;
        }
    } else if (prev.getSpan () .intersects (curr.getSpan ())) {
        if (prev.getProb () > curr.getProb ()) {
            deleteCurr = true;
            break;
        } else {
            allAnnotations.remove (stack.remove (ki)) ;
            ai--;
        }
    } else if (prev.getSpan () .contains (curr.getSpan ())) {
        break;
    } else {
        stack.remove (ki) ;
    }
}
if (deleteCurr) {
    allAnnotations.remove (ai) ;
    ai--;
    deleteCurr = false;
} else {
    stack.add (curr) ;
}
}
}

```

for 循环结束时在
deletion to negate
ai++ 后更新下标

对栈中的每个元素循环

测试两个命名实体 Span
是否相等，如果相等则
去掉概率小的那个

测试两个命名实体 Span
是否相等，如果相等则
去掉概率小的那个

for 循环结束时在
deletion to negate
ai++ 后更新下标

测试某个实体 Span 是否
已经被另一个 Span，
如果是则退出循环

测试某个实体 Span 是否
已经被另一个 Span，
如果是则退出循环

测试某个实体 Span 是否在另一
个实体 Span 之后，如果是则从
栈中剔除前面的命名实体

这种合并方法的时间复杂度对于句长来说是线性的，但是由于允许命名实体出现在另外的命名实体中，上面使用第二个循环来处理保留嵌套命名实体的栈。该栈的大小永远不会超过所用的命名实体类型数目，因此第二个循环的复杂度可以看成常数。到现在为止，我们讨论了一些命名实体识别的背景知识，并介绍了一个使用多模型的例子，接下来深入考察利用 OpenNLP 实现实体识别的工程细节。

5.3.2 OpenNLP识别实体的背后机理

如果问一个非工程师“电视是怎么工作的？”，他可能会回答：“哦，你拿遥控器点一下它，然后按一些红色按钮……”。到现在为止，我们只是介绍如何使用OpenNLP中的命名实体识别软件，这相当于提供和上例一样的回答。本节将考察该软件在识别命名实体时实际的运行原理。在5.4节和5.5节讨论性能和定制相关话题时，这些信息就很有价值。

OpenNLP将命名实体识别看成是第7章将要介绍的标注任务的一种。该过程为每个词条标记如下三种标签之一。

- 开始：该词条是新的命名实体的开始。
- 继续：将已有的命名实体扩展到该词条。
- 其他：该词条不是命名实体的一部分。

对于一个典型的句子，该标注过程类似表5-1。

表 5-1 为识别命名实体而标注的句子

0	1	2	3	4	5	6	7	8	9	10	11
“	It	is	a	familiar	story	,	“	Jason	Willaford	Said	.
other	other	other	other	other	other	other	other	start	continue	other	other

通过把“开始”标签和任意数目的“继续”标签连接起来，上述分类结果序列就可以转换成Span集合。OpenNLP所使用的统计建模包通过构建模型来确定每一个标签的预测结果。该模型使用代码中指定的特征集合来预测最有可能的结果。这些特征可以区分专有名词、不同的数字字符串类型、词周围的上下文以及标注结果。OpenNLP中使用的命名实体识别特征如下。

- 1 待标注词条
- 2 左侧第一个词条
- 3 左侧第二个词条
- 4 右侧第一个词条
- 5 右侧第二个词条
- 6 待标注词条的词类
- 7 左侧第一个词条的词类

- 8 左侧第二个词条的词类
- 9 右侧第一个词条的词类
- 10 右侧第二个词条的词类
- 11 待标注词条及其词类
- 12 左侧第一个词条及其词类
- 13 左侧第二个词条及其词类
- 14 右侧第一个词条及其词类
- 15 右侧第二个词条及其词类
- 16 左侧第一个词条预测的结果或者NULL
- 17 左侧第二个词条预测的结果或者NULL
- 18 待标注词条及左侧第一个词条
- 19 待标注词条及右侧第一个词条
- 20 待标注词条的词类及左侧第一个词条的词类
- 21 待标注词条的词类及右侧第一个词条的词类
- 22 先前分配给当前词条字符串的结果或者NULL

很多特征都基于待标注词条及其邻近词条，但是有些特征基于词类。词条的词类源自词条的基本性质，比如是否只由小写字符构成。

这些特征基于所用的词、这些词的类别集合及其在当前文档或句子中先前做的决策，来对构成实体的词条以及实体的类型进行建模。词本身十分重要，它们替代了规则方法中的列表。如果训练数据包含某个被足够多次标注为某个实体的词，那么利用特征1的分类器能够简单地记住那个词。和特征1类似，特征6集中关注待标注的词，但是这里使用的不是词本身而是其词条类别。

用于命名实体识别的词条类别如下：

- 1 词条由小写字母构成
- 2 词条是两个数字
- 3 词条是四个数字
- 4 词条包含一个数字和一个字母
- 5 词条包含一个数字和一个连字符
- 6 词条包含一个数字和一个反斜杠

7 词条包含一个数字和一个逗号

8 词条包含一个数字和一个句号

9 词条包含一个数字

10 词条全部大写，单字母

11 词条全部大写，多字母

12 词条首字母大写

13 其他

上面列出的词条类别用于帮助预测某些实体类型。例如，词条类别3表明是一个年份，如1984。而词条类别5和6也是典型的日期表示，而类型7和8是更典型的货币量的表示。特征2到15可以将词所在的上下文考虑在内，这样像*Washington*一样的歧义词在上下文“in Washington”中就更可能被识别成地名，而如果出现在“Washington said”中则更可能被识别为人名。特征16和17能够允许“继续”标签紧跟“开始”标签，而特征18能够获得单词是否先前被标记为人名实体的一部分这个信息，如果是的话那么同样的词在后面提到时也可能是人名的一部分。尽管所有特征自己都不能进行完整预测，但是对这些特征的进行经验性的加权组合之后就能够得到OpenNLP中的命名实体类型。

上述特征的目标是获得OpenNLP中所识别的实体类型所需的信息类型。对于你自己的应用而言，可能存在这样的情况，即OpenNLP所提供的实体类型已经能满足需求。由于命名实体的自动识别技术永远都不会可能完美（人工方法亦是如此），对于你的应用来说问题就是识别的精度是否足够？在下一节中，我们会考察OpenNLP中所带的模型在识别这些实体时到底能够达到何种性能，我们还会考察软件的实际运行性能。这些特性能够帮助你确定是否能够使用该现成软件，以及能够使用这些软件的应用类型如何。在你的应用中，可能要识别其他的实体类型。这种情况下，这里列出的特征可能不能刻画对新实体类型建模所需要的信息。

5.4 OpenNLP的性能

我们将考虑与命名实体识别相关的三种领域的性能。第一是语言标注的质量，或者更明确地说就是OpenNLP模块发现的命名实体的质量。正如前面所讨论的那样，语言分析永远不会完美，但是这并不会阻碍我们考察系统和人之间的匹配近似

程度。利用这个信息，就可以知道所提供模型的精度对你的应用而言是否已经足够，或者是否需要额外的训练数据或定制处理。

要考虑的第二种性能是任务运行的速度。我们会讨论OpenNLP所做的一些优化处理措施，这些措施可以提高运行时的效率，并能对多个模型应用于文本的速度进行评估。最后，我们会考察命名实体识别所需的内存大小。如前所述，命名实体识别模型可以看成相互分离的模型从而只需要使用那些需要的模型。这样做的部分原因是因为命名实体模型需要大量内存。我们会考察所需要的精确内存大小，同时考察一种能够大大降低后续模型所需内存的方法。所需内存信息可以确定该技术是否适用于你的应用，特别是可以确定命名实体识别技术到底是以在线还是离线方式运行。

5.4.1 结果的质量

命名实体识别系统的语言质量取决于训练所用的数据。OpenNLP使用的命名实体识别模型构建于MUC-7 (http://www-nlpir.nist.gov/related_projects/muc/proceedings/muc_7_toc.html) 任务的数据之上。该数据集常常用于研究上的命名实体识别系统。该任务对于人名、机构名、地名或其他实体类型的定义给出了十分具体的标准。尽管这些实体类型看上去十分清晰，但是在实际文本中有时并不那么明显，需要指导方法来解决。例如，大部分有名称的人工产品（如Space Shuttle Discovery——发现号航天飞机）都不会给分类，但是按照指导准则，机场都要看成地名。在这些类别的识别学习过程中，对特定的边界样例的精确选择结果不是特别重要，重要的是这些样例的标注必须一致，这样模型才能对它们进行学习。

我们利用MUC任务中的训练集和测试集对OpenNLP中带的模型进行评估。OpenNLP不是在测试集数据上进行训练的，因此它能够提供未知文本上的合理性能评估结果。评估的结果如表5-2所示。

表 5-2 OpenNLP 所产生的的标注的质量评估结果

数 据 集	正 确 率	召 回 率	F 值
MUC-7 训练数据集	87	90	88.61
MUC-7 测试数据集	94	75	83.481

正确率给出的是系统识别结果中正确的比例，而召回率给出的是到底有多少真

正的命名实体被找出来的比例。F值是正确率和召回率的加权调和平均值。从表中结果可以看出，OpenNLP能够识别至少75%的命名实体，而错误率只在大约10%左右。

5.4.2 运行性能

第二类要考虑的性能是运行时的性能。在5.1.2节我们讨论过，识别模型将句子中的每个词标注为“开始”、“继续”或“其他”三种标签。这意味着对需要检测的每种实体类型，模型必须要对每个词条进行决策。对于所处理文本的任一单位来说，最多有三种可能的实体集合要考虑，因此决策的潜在数目要乘以3。当模型加入系统的时候，开销就会变得过大。

OpenNLP通过两种方式来减轻这种处理开销。首先，对于结果概率进行缓存处理，当用于预测某个结果的特征对三种实体类型都一模一样时，该概率分布只需要计算一次然后就进行缓存处理。其次，特征生成本身进行缓存处理。由于所有模型使用相同的特征，而每次只处理一个句子，不依赖于模型上次决策的句子级特征，只需计算一次并缓存结果。这样做的结果是，尽管使用更少的模型明显快于使用更多的模型，在模型增加时开销并不严格线性增长。图5-2给出的性能图展示了模型增加时其运行性能并没有按照严格的线性方式下降。

在5.5节中，我们会讨论另一个放弃了一些灵活性的模型，它能够检测所有实体类型，但是其运行性能却与单个模型相当。

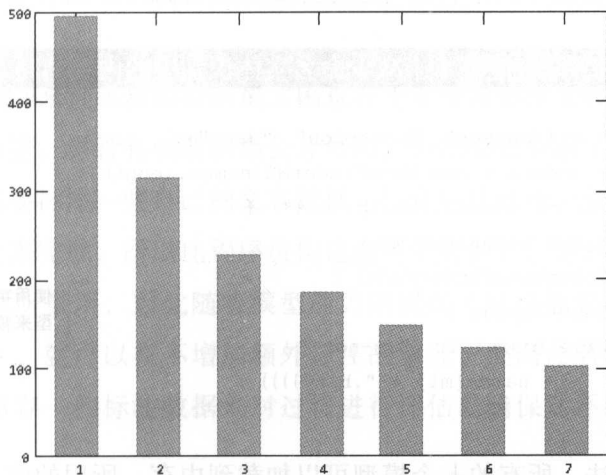


图5-2 命名实体工具每秒处理的句子数目和识别的模型类型数目的关系

5.4.3 OpenNLP的内存使用

本节将考察OpenNLP中命名实体识别系统的内存要求。正如在前一节看到的那样，维护独立的模型会带来一些开销。当使用人名识别模型时，单个进程大概需要消耗68M内存，除去代码和JVM消耗的内存外，模型消耗的内存大概为54M。模型由特征、结果及参数值构成，但是大部分空间主要用于存储特征的名称。这是因为我们使用了词汇特征，因此训练数据中出现超过一定次数的每个词会在我们的模型中以特征的方式存储。而由于词汇特征可能与其他词汇特征和非词汇特征组合，模型所需要存储的特征总数可能很大。将所有模型加载到内存大概需要400M内存。

前一节提到，每个命名实体模型都使用相同的特征集合。如果它们都基于相同数据训练，那么包含的特征集也一样，只是由于不同的因素对不同的实体类型更重要，因此上述特征参数会有所不同。即使这些模型不在同一数据集上训练得到，由于不同文本片段通常会包含很多相同词，因此特征之间的重合度也会很高。如果某种程度上能够共享这些特征所分配的内存，就可以期望在使用多个模型时显著减少内存的使用。一种实现这种做法的机制是Java的String.intern () 方法。该方法通过实现一个字符串池返回字符串的规范表示。利用这种方法，可以保证所有某个特定字符串的引用都指向内存中的同一个对象。

本书所附代码中包含了一个模型读取器，该读取器使用String.intern () 方法来实现上述效果。清单5-7对刚才那个多模型的例子重新进行考察，以理解该模型读取器的用法。

清单5-7 利用字符串池技术来减少命名实体识别中的内存消耗

```
String[] names = {"person", "location", "date"};
NameFinderME[] finders = new NameFinderME[names.length];
for (int mi = 0; mi < names.length; mi++) {
    finders[mi] = new NameFinderME (
        new PooledTokenNameFinderModel (
            new FileInputStream (
                new File (modelDir, "en-ner-"
                    + names[mi] + ".bin"))));
}
```

对识别人名、地名和日期的命名实体识别器进行初始化

使用字符串池模型来减少内存量

利用上述方法，所有的七个模型可以加载到内存，所用的内存量约为 225M，

大概节省了 175M 内存。由于将特征映射为公共表示在模型加载时完成，因此将此模型应用于文本时不会影响其运行性能。

现在我们理解了有关 OpenNLP 质量、速度和内存使用的基本知识，下面从另外一个层次进行探讨，看看如何为自己的领域定制 OpenNLP。下一节将考察在部署自己的应用时如何训练自己的模型并做其他定制处理。

5.5 对新领域定制 OpenNLP 实体识别

很多情况下，OpenNLP 提供的模型对于你的应用或领域来说已经足够。但是有些情况下你可能需要训练自己的模型。本节将讨论如何利用 OpenNLP 来训练自己的模型，以及如何改变所用的特征来预测实体，并且将介绍另一种使用 OpenNLP 来识别实体的方法，该方法在某些情况下具有一定的优势。

5.5.1 训练模型的原因和方法

训练自己的模型有很多原因。例如，你可能需要识别一种新的实体类型，比如车辆。或者，你可能需要识别 OpenNLP 所提供的人名或其他实体类型，但是你所面对的领域迥然不同，这使得 OpenNLP 的处理效果不够好。或者有些特殊情况下你需要采用与 OpenNLP 模型不同的人名定义。此外，如果正在识别不同的实体类型或者在不同领域使用，你可能有一些新特征在模型中使用。最后，OpenNLP 使用的切词方法可能不适合于你的领域或者后续处理过程，这种情况下需要使用一个新的切词器来训练模型。

利用 OpenNLP 来训练新模型的最大困难在于发现或创建足够的对于统计建模实际可用的训练数据。尽管有些数据集公开可用，但如果要为新实体类型构建模型，那么几乎肯定需要标注一些自己的文本数据。尽管标注过程十分耗时，但是通常并不需要专业人士来完成，所以比程序员构建规则集要便宜。另外，构建的数据集也能被不同的模型所重用，因此随着模型能力的提高（或许能够利用更具预测性的特征进行识别），就可以在不增加额外标注的情况下提高命名实体识别系统的性能。你也必须留存一些标注数据来对过程进行评估以确保对系统的修改能提高整体的性能。

如果寻求在新领域中改善 OpenNLP 随带的模型性能，就有更多的做法。尽管

OpenNLP发布了一个模型集合，由于许可证限制的原因，它并没有发布构建这些模型的训练数据。对于这个问题来说，三种常见的解决方法如下。

- 只使用领域数据：利用足够的数据，有可能产生最准确的面向领域的模型
- 构建一个独立的模型并组合结果：这与本节前面看到的方法类似，在那里我们也组合了不同类型的标注，只有在这种情况下，两个分类器都预测相同的类别。如果两个分类器的正确率都很高，那么组合它们的结果会有助于提高召回率。
- 利用OpenNLP模型的输出来训练数据：这种方法能够提高可用的训练数据量。如果和一些人工纠错一起用会更好。OpenNLP模型是在新闻文本上训练得到的，因此将它应用于相似文本会得到最好的结果。

不管需要定制OpenNLP的具体情况如何，下面几节的内容都有助于理解该过程的执行方式。

5.5.2 训练OpenNLP模型

现在你已经了解训练新命名实体识别模型的原因和时机，并且你已经拥有了一些标注数据，下面看看如何进行训练。OpenNLP提供了在NameFinderME.main () 中训练的代码，能够支持一些选项，比如指定字符集和其他一些特征。在清单5-8中，我们将看到上述代码的一个精简版本，其中的代码也稍微进行了整理。

清单5-8 在OpenNLP中训练命名实体模型

```
File inFile = new File (baseDir, "person.train") ;
NameSampleDataStream nss = new NameSampleDataStream (
    new PlainTextByLineStream (
        new java.io.FileReader (inFile))) ;
int iterations = 100;
int cutoff = 5;
TokenNameFinderModel model = NameFinderME.train (
    "en", // language
    "person", // type
    nss,
    (AdaptiveFeatureGenerator) null,
    Collections.<String, Object>emptyMap (),
    iterations,
    cutoff) ;
```

← 基于标注数据为
实体样本构建流

← 训练模型

```
File outFile = new File (destDir, "person-custom.bin") ;
FileOutputStream outFileStream = new FileOutputStream (outFile) ;
model.serialize (outFileStream) ;
```

← 将模型存到文件中

上述代码的开始两行指定包含训练数据的文件，并创建了一个NameSampleStream。NameSampleStream是一个十分简单的接口，它能允许在一系列NameSample上进行迭代，其中NameSample是一个简单的保存命名实体 Span 和词条的数据结构。NameSampleDataStream实现了上述接口，每行分析一个句子。每行包括空格分隔符，其中实体通过空格分开的<START>和<END>标签来标记：

```
"It is a familiar story " , <START> Jason Willaford<END> said .
```

尽管提供了对上述格式的支持，其他格式也可以很容易通过编写一个实现NameSampleStream的类来分析完成。

训练例程有多个参数。前两个参数给出的是要产生模型的语言和类型。下一个是用于生成NameSample的NameSampleDataStream，而这些NameSample会转换成用于训练模型的事件流。正如在5.1.2节看到的那样，OpenNLP会将每个实体看成是基于上下文的一系列“开始”/“继续”/“其他”的决策结果。

训练方法的下一个参数是包含训练参数的一个对象，它由ModelUtil.createTrainingParameters来构造。其封装了迭代的数目以及用于模型创建的特征截断值。迭代参数可以很大程度上被忽略，但是当模型训练时，它会输出100次迭代的每一步输出。特征截断参数提供了包含在模型当中的特征必须要出现的次数下界。该参数的默认值为5，即所有出现少于5次的特征不会包含在模型中。由于对仅出现几次的特征模型无法精确估计其参数，所以对于控制模型的规模以及可能的噪声来说，上述做法是必须的。该值设置太低的话会导致模型在未知数据上效果很差。但是对于小数据集，该截断参数值意味着，如果样本中单词出现次数少于5次，那么样本上下文的预测性就不是很好，模型可能就无法对训练数据中的样本正确分类。

下一个参数是placeholder。AdaptiveFeatureGenerator参数赋空的话会导致NameFinderME使用对于命名实体识别很有效的特征生成器的默认集合。没有额外的资源添加到生成模型中，因此对于资源参数使用了一个空map。

代码的最后一行将模型写到磁盘中。文件名表明模型应该写到一个二进制压缩

文件中。尽管opennlp.maxent包也支持其他的格式，这种格式才是这段将模型应用到新文本的代码所期望的格式。

5.5.3 改变建模输入

迄今为止，我们已经讨论了训练自己模型的多个原因，也给出了训练模型的基本知识。如前所述，其中两个原因分别涉及修改建模过程的输入以及获取或构建自己的标注集。我们考察第一类修改涉及切词过程。

修改切词过程涉及多个步骤。首先，在训练过程中，不管原始格式如何，训练和测试文本必须要转换为类似于上一节Jason Willaford例子展示的空格分隔的词条形式。尽管这种转换可以按照你的要求完成，我们建议使用第二个过程即在未知文本上应用模型相同的代码基。对于在未知文本上应用模型，又是一件识别新词条并将它们输送给NameFinderME.find方法的事情。该步骤涉及编写自己的OpenNLPTokenizer类的实现代码，这一点与5.2节使用的opennlp.tools.tokenize.SimpleTokenizer类似。由于继承该类只涉及分割String并返回String数组，所以这里不再展示例子而是直接转到改变所使用的特征。

我们要考虑的训练和测试例程输入的其他改变，是修改模型所使用的特征来预测实体。同切词类似，修改用于识别实体的特征同样十分直观，NameFinderME类配置为接受一个AggregatedFeatureGenerator即可，它可以配置成包含多个特征generator构成的集合，如下所示。

清单5-9 OpenNLP为命名实体识别中生成定制特征

```

AggregatedFeatureGenerator featureGenerators =
    new AggregatedFeatureGenerator (
        new WindowFeatureGenerator (
            new TokenFeatureGenerator (), 2, 2),
        new WindowFeatureGenerator (
            new TokenClassFeatureGenerator (), 2, 2),
        new PreviousMapFeatureGenerator ()
    );

```

创建 5 个词条窗口内词条的特征生成器（左边 2 个词条，右边 2 个词条）

生成包含下面定义的 3 个生成器的聚合特征生成器

创建 5 个词条窗口内词条类别的特征生成器（左边 2 个词条，右边 2 个词条）

创建指定该词条先前标注结果的特征生成器

OpenNLP 中包含了大量不同的 AdaptiveFeatureGenerator 实现以供选择，或者你可以很容易地实现自己的特征生成器。下面给出了一些可用的类及其用途。

- CharacterNgramFeatureGenerator: 使用字符 n 元组来生成每个词条的特征。
- DictionaryFeatureGenerator: 如果词条包含在词典中则生成特征。
- PreviousMapFeatureGenerator: 生成与先前出现的词有关的结果的特征。
- TokenFeatureGenerator: 生成包含词条自己的特征。
- TokenClassFeatureGenerator: 生成反映词条不同方面的特征, 这些方面包括字符类型(数字/字母/标点符号)、词条长度和大小写等。
- TokenPatternFeatureGenerator: 基于字符类型将词条分割成子词条, 并对每个子词条及其组合生成类型特征。
- WindowFeatureGenerator: 对给定的AdaptiveFeatureGenerator生成跨窗口内词条的特征(比如左侧第一个词条、右侧第一个词条)。

在训练例程中, 需要修改对NameFinderEventStream类的调用以便也包含一个定制的名称ContextGenerator类构造函数, 如下所示。

清单5-10 在OpenNLP中利用定制特征来训练命名实体模型

```
File inFile = new File (baseDir, "person.train") ;
NameSampleDataStream nss = new NameSampleDataStream (
    newPlainTextByLineStream (
        new java.io.FileReader (inFile))) ;

int iterations = 100;
int cutoff = 5;
TokenNameFinderModel model = NameFinderME.train (
    "en", // language
    "person", // type
    nss,
    featureGenerators,
    Collections.<String, Object>emptyMap (),
    iterations,
    cutoff) ;

File outFile = new File (destDir, "person-custom2.bin") ;
FileOutputStream outFileStream = new FileOutputStream (outFile) ;
model.serialize (outFileStream) ;
```

构建样本流

利用定制的特征生成器训练模型

将模型存到文件中

类似地, 对于测试而言, 修改对NameFinderME类的调用以便也包含一个定制的名称ContextGenerator类构造函数, 如下所示。

清单5-11 在OpenNLP中利用定制特征来使用命名实体模型

```

NameFinderME finder = new NameFinderME (
    newTokenNameFinderModel (
        newFileInputStream (
            new File (destDir, "person-custom2.bin")
        )), featureGenerators, NameFinderME.DEFAULT_BEAM_SIZE);

```

探讨如何修改OpenNLP的训练机制之后，现在就可以对新的实体类型建模并获取所要检测实体的新信息类型。这可以让你将软件扩展到广阔的其他实体类型和文本领域中去。甚至得到这些信息之后，在某些情况下，OpenNLP可以在如何以多大内存和运行性能开销之间做出取舍，从而具有一定的灵活性。下一节将探讨进一步定制OpenNLP命名实体软件来实现性能的改进。

5.5.4 对实体建模的新方法

前面提到，OpenNLP对每种实体类型构建一个单独的模型，这样就可以允许用户灵活地选择想要使用的模型。本节将探讨一种实体建模的新方法，它可能没有那么灵活，但是有其他的一些优点。前面我们看到，可以对每个词条预测其属于三种结果之一（开始、继续或其他）来完成实体的建模。这里我们考虑一种同时包含待识别实体类型的模型。利用这种方法，我们就可以预测出类似人名一开始、人名一继续、日期一开始、日期一继续、其他等结果，具体的结果类型取决于要模型完成的预测任务本身。表5-3给出了一个句子的预测结果。

表 5-3 利用另一个命名实体识别模型对句子的标注结果

0	1	2	3	4	5	6	7	8	9
Britney person-start	Spears person-continue	was other	reunited other	briefly other	with other	her other	sons other	Saturday date-start	. other

相比于每种实体类型采用独立模型的方法来说，这种方法有多个独特的优点，但同时也带来一些局限性和缺点。

该方法的优点如下。

- 可能会减少运行时间：由于只使用单个模型，特征只需要对所有实体类型计算一次。同样，对于该模型只需要计算一个预测结果集。处理句子时，所考

察的候选集数目可能随着结果数目的增长而增长，但是不太可能需要对每种实体类型需要计算三个集合。

- 可能会节省内存：对于模型来说，只需要将一个特征集合加载到内存。此外，因为`other`这个标签会被多个实体类型所共享，所以参数也会更少。
- 不需要实体合并：在一个模型下，不需要进行实体合并。

该方法的缺点如下。

- 不支持实体类型的重叠：由于对每个词条只赋予一种标签，因此不支持命名实体的嵌套。这个问题通常可以通过在训练集上只标注区间最长的命名实体来解决。
- 有可能损失内存、运行性能：由于只存在单个模型，因此无法选择使用所学要的模型的一部分。这可能会导致内存和运行时间的损失，特别在那些只需要一个或两个实体类型的情况下更是如此。
- 训练数据：无法利用对所有类型已经进行标注的训练数据。在本模型中增加新的实体类型需要对所有训练数据进行标注。

在各种取舍情况下，该方法是否有效将取决于应用的需求。有可能两类模型的组合是应用中的最佳使用模式。5.3.1节中介绍的组合单个模型方法也适用于将这两类模型组合成单个模型。

为构建这种模型，需要做一些修改。首先训练数据需要反映所有的标注而不是单个标注类型。这可以通过将原来的训练数据转换成支持多种标签类型的格式来实现。一个具体的例子如下。

```
<START:person> Britney Spears <END> was reunited with her sons <START:date>
Saturday <END>.
```

采用这种格式的训练数据，可以用类似5.5.2节的方法来使用OpenNLP提供的NameSampleDataStream类。下面的清单展示了产生该模型的过程。

清单5-12 训练具有不同实体类型的模型

```
String taggedSent =
    "<START:person> Britney Spears <END> was reunited " +
    "with her sons <START:date> Saturday <END> ";
ObjectStream<NameSample>nss = new NameSampleDataStream (
```

```

newPlainTextByLineStream (new StringReader (taggedSent))) ;
TokenNameFinderModel model = NameFinderME.train (
    "en",
    "default" ,
    nss,
    (AdaptiveFeatureGenerator) null,
    Collections.<String, Object>emptyMap (),
    70 , 1 ) ;

File outFile = new File (destDir, "multi-custom.bin") ;
FileOutputStream outFileStream = new FileOutputStream (outFile) ;
model.serialize (outFileStream) ;
NameFinderME nameFinder = new NameFinderME (model) ;
String[] tokens =
    (" Britney Spears was reunited with her sons Saturday .")
    .split ("\\s+") ;
Span[] names = nameFinder.find (tokens) ;
displayNames (names, tokens) ;

```

该模型将“开始”和“继续”的结果映射为给定类型的名称。实体的类型预先考虑了结果中的标签名称。人名一开始和人名一继续标签序列产生了指定人名的词条Span，类似地，日期一开始、日期一继续标签指定日期。当利用模型来处理输入时，通过getType ()方法来访问每个产生的Span的实体类型。

5.6 小结

在涉及文本处理的应用中，识别其中的专有名词并对它们分类可以带来十分丰富的信息。我们讨论了如何利用OpenNLP来识别实体，并提供了结果质量、内存需求量、处理速度等指标来度量识别的性能。我们也考察了如何训练自己的模型并探讨了这样做的原因。最后，我们介绍了OpenNLP是如何来完成这个任务的，同时介绍了定制模型的方法，并考虑了另一种对实体建模的方法。这些主题可以让你在文本处理应用中利用高性能的命名实体识别方法。此外，它们为分类系统在文本处理中如何使用提供了一些感觉。在本书后面考察分类时，我们会重新讨论这个主题并进行深入的探讨。接下来，我们将考察如何利用一种称为聚类的技术将相似项（如整篇文档和搜索结果）自动分组。与分类不同，聚类通常是一个无监督的任务，这也意味着它不需要训练模型，但是能够按照某种相似度计算方法自动将对象归类。

5.7 进一步阅读材料

Mikheev, Andrei; Moens, Marc; Glover, Claire. 1999. "Named Entity Recognition without Gazetteers." Proceedings of EACL '99. HCRC Language Technology Group, University of Edinburgh. <http://acl.ldc.upenn.edu/E/E99/E99-1001.pdf>.

Wakao, Takahiro; Gaizauskas, Robert; Wilks, Yorick. 1996. "Evaluation of an algorithm for the recognition and classification of proper names." Department of Computer Science, University of Sheffield. <http://acl.ldc.upenn.edu/C/C96/C96-1071.pdf>.

Zhou, GuoDong; Su, Jian. 2002. "Named Entity Recognition using an HMM-based Chunk Tagger." Proceedings of the Association for Computational Linguistics (ACL), Philadelphia, July 2002. Laboratories for Information Technology, Singapore. <http://acl.ldc.upenn.edu/acl2002/MAIN/pdfs/Main036.pdf>.

第6章 文本聚类

本章内容

- 常见文本聚类算法的基本概念
- 聚类如何提升文本应用的例子
- 如何对词进行聚类来识别感兴趣的主体
- 使用Apache Mahout对文档集聚类，使用Carrot²对搜索结果聚类

你可能有这样的频繁经历：浏览在线文章时，会点击一个感兴趣的标题，但是却发现点进去的内容和你刚刚看完的基本一样。或者你可能面临的任务是向老板简要汇报一天的新闻，此时需要的只是一个摘要和几个关键点，但是你却没有时间通读所有的内容。或者你的用户常常输入一些有歧义的或一般性的查询，或者你的数据覆盖了很多主题而你想将搜索结果分组以减少用户阅读不相关结果所花的时间。采用自动将相似项分组并以摘要方式展示结果的自动化工具是一个很好的方式，这样不需要阅读所有或大部分内容就可以通读大量文本或搜索结果。

本章将近距离考察上述问题的解决办法，其中之一是使用一种称为聚类的机器学习方法。聚类是一种无监督的任务（不需要像标注训练语料一样的人工干预），它能够自动将相关内容放到相同桶中，这样就可以更好地组织内容或者减少需要人工处理的内容量。在某些情况下，聚类方法同时也可以为每个桶分配标签甚至给出每个桶的内容摘要。

本章第一节将了解聚类的概念，之后我们将深入考察如何利用一个称为Carrot²的项目来对搜索结果聚类。然后，我们将考察如何利用Apache Mahout来对大规模文档集聚类。实际上，不论是Carrot²还是Mahout都带了多种聚类方法，每种方法都有自己的优缺点。我们还将通过一个称为LDA（Latent Dirichlet Allocation）的技术来考察词粒度上的聚类以识别文档中的主题（该任务有时称为主题建模），在Apache Mahout中恰好也实现了这个模型。通过这些例子，我们会展示如何将这些做法构建到前面Apache Solr的例子中，以通过各种访问路径更加容易地访问簇信息，同时也容许对信息更丰富的访问。最后，我们将以性能问题结束本章的讨论，同时考虑数量（多快？）和质量（多好？）。首先，先看一个大家可能耳熟能详的例子Google News，但是大家可能并不知道它是基于聚类实现的。

6.1 Google News中的文档聚类

在24小时新闻周期内，数不尽的新闻渠道发布了他们关于事件的不同版本，而Google News能够将相似报道聚类，从而允许读者快速阅读特定时间段内发布的有关某个主题的新闻报道。例如，在图6-1中，头条新闻“Vikings Begin Favre era on the road in Cleveland”显示有2181篇其他报道与主报道相似。尽管不清楚Google实现上述功能的具体聚类算法，Google的文档却清楚表明该功能是通过聚类来实现的（Google News 2011）：

我们的分组技术考虑了很多因素，比如标题、文本和发布时间等。然后我们使用多种聚类算法来识别密切相关的报道。Google News上展示的报道给出了新闻报道、视频、图像和其他一些信息。

通过互联网的连接，这种大规模下分组的威力对任何人都显而易见。尽管对于近实时新闻内容分组来说，不仅仅是对内容进行聚类，但是设计成像Apache Mahout那样规模的聚类实现对于起步来说至关重要。

在类似新闻聚类的任务中，应用必须能够快速对大量文档进行聚类，确定代表性文档或者标签用以展示，并能处理新来的文档。对于该问题来说不仅仅是拥有一个好的聚类算法那么简单，但是为达到我们的目标，我们将集中关注聚类如何能够解决这些或其他能够发现和处理信息的非监督任务。

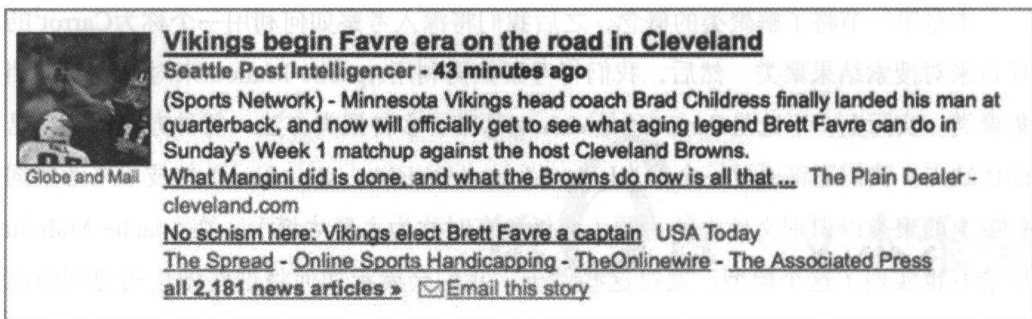


图6-1 Google News对新闻进行聚类的例子，截图于2009年9月13日

6.2 聚类基础

聚类是基于某种相似度计算方法将未标记文档分组的过程。其目标是将文档集的所有文档分成多个簇，簇中文档之间相似，而不相似文档出现在不同簇中。在了解基础知识之前，大体上设定聚类的预期很重要。尽管聚类很重要，但它不是包治百病的万能药。聚类体验的质量往往归结于对用户设定的预期。如果用户期望完美，那么他们会失望。如果他们期望聚类能够在很大程度上帮助自己快速通读大量文档，同时仍然需要处理伪正例，那么他们可能会感到比较满意。基于应用设计者的角度来看，需要相当规模的测试来寻找执行速度和结果质量之间折中的正确设置。根本上来说，记住你的目标是鼓励发现并偶尔与内容进行交互，并不一定有完美的相似对象。

介绍基本的定义和可能的警告之后，接下来有关基础知识的各小节中将考察：

- 能够应用聚类的不同类型文本。
- 如何选择聚类算法。
- 确定相似度的方法。
- 确定簇标签的方法。
- 如何评估聚类结果。

6.2.1 三种聚类的文本类型

可以对不同形式的文本进行聚类，包括文档中的词、文档本身或者搜索的结果。聚类对很多除文本之外的其他对象来说也十分有用，但是这些已经超出本书的

范围。现在，我们集中关注三种类型的聚类：文档、搜索结果及词/话题。

在文档聚类中，主要像前面Google News的例子一样将文档作为整体进行分组。文档聚类通常作为一个离线批处理作业来完成，其输出结果通常是一个文档列表和一个中心向量。由于文档聚类通常是一个离线处理任务，因此值得花额外的时间（在合理的范围内）来获得更好的效果。对于簇的描述通常通过考察离中心最近的文档中最重要的词项（由某种权重机制来确定，如TF-IDF）来生成。通常需要一些预处理过程来去除停用词和进行词干还原处理，但是这些处理不一定对所有算法来说都是必须的。其他常见的文本技术，如短语识别或使用 n 元组等可能也值得在对方方法进行测试时进行反复试验。为了解更多有关文档聚类的知识，请从*An Introduction to Information Retrieval*（Manning 2008）开始。

对于搜索结果聚类，在给定用户查询的情况下，聚类任务在搜索后将搜索结果聚成多个簇。当用户输入一般性或有歧义的词项（如apple）或当数据集包含不同类别时，搜索结果聚类可能会十分有效。搜索结果聚类常常有如下几个特点。

- 对短文本片段（标题，或许和查询词项匹配的一部分正文文字）进行聚类；
- 算法设计成在小规模结果集合上运行并尽可能快地返回结果；
- 标签十分重要，这是因为用户可能会将簇看成面，以确定如何进一步浏览结果集。

在搜索结果聚类中，往往和文档聚类一样进行预处理，但是由于标签通常更加重要，因此花额外时间来识别常见短语是有意义的。一篇有关搜索结果聚类的综述请参考“A Survey of Web Clustering Engines”（Carpineto 2009）。

将词聚成主题，也称为主题建模，是一种高效地从大量文档集合中找出它们所讨论主题的方法。该方法基于这样一个基本假设：即文档常常覆盖多个主题，而与某个给定主题相关的词互相邻近。通过词聚类，可以快速发现哪些词互相靠近，然后也发现哪些文档与这些词相关。（从某种意义上说，该方法也进行了文档聚类。）例如，运行一个主题建模算法（请参考6.6节）后的结果会产生类似表6-1的词聚类结果（基于显示的原因，这里对原始结果进行了格式化处理）。

表 6-1 主题和词的例子

主题 1	主题 2
win saturday time game know nation u more after two take over back has from texa first day man offici 2 high one sinc some sunday	yesterday game work new last over more most year than two from state after been would us polic people team run were open five american

在本例中，需要注意的第一件事是主题本身缺乏名称。对主题命名是生成主题的人所做的工作。第二件事是你甚至不知道哪篇文档包含哪个主题。为什么要自找麻烦呢？生成文档集的主题是另外一种辅助用户浏览文档集并在不需要阅读所有文档集就可以发现有趣信息的方法。此外，最近一些工作可以通过短语来更好地刻画主题（参考Blei [2009]）。

为学习更多有关主题建模的知识，可以参考本章最后列出的参考文献，也可以访问http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation，该页面提供了有关主题建模的主要学术论文。

现在已经对要聚类的文本类型有了基本的了解，下面就看看选择聚类算法时要考虑的因素。

6.2.2 选择聚类算法

有很多不同的聚类算法可供选择，介绍所有算法不在本书的范围之内。例如，在本书写作之时，Apache Mahout包含了K-Means（会在后面介绍）、模糊K-Means、Mean-Shift、Dirichlet、Canopy、Spectral及LDA的实现，毫无疑问到本书出版时其包含的算法会更多。我们不会深入探索每种算法的实现过程，而是看看聚类算法的一些共同特征，以便更好地理解在选择聚类算法时有用的准则。

在讨论聚类算法时，需要进行多方面的考察以确定与应用最吻合的算法。传统上来说，一个主要的决定因素是算法本质上到底属于层次算法还是扁平算法。正如名称暗示的那样，层次方法可以自顶向下或自底向上运行，相关文档可以按照层次方式组织，即可以继续划分为越来越小的集合。而扁平方法因为不必将不同簇联系起来，因此其速度通常要快很多。要记住的是，有些扁平算法可以修改成层次算法。

除层次和扁平之外，表6-2给出了其他一些用于选择聚类方法的因素。

表 6-2 聚类算法的选择

特 性	描 述
簇隶属关系（软 / 硬）	硬——文档属于且仅属于一个簇 软——文档可以属于多个簇，通过概率来表示属于每个簇的隶属度
可更新	当新文档到来时簇能否更新还是需要重做全部计算？
概率方法	理解方法的支撑基础将有助于了解这种方法的优缺点
速度	大部分扁平算法的运行时间与文档数目呈线性关系，而很多层次方法是非线性的
质量	层次方法通常比扁平方法要精确，当然时间开销要更大一些。更多的评估参考 6.2.5 节
反馈处理	算法能否基于用户的反馈调整 / 提高？例如，如果用户标记某篇文档不适合某个簇，那么算法能否排除这篇文档？这样做会不会改变其他簇？
簇的数目	有些算法需要应用来预先决定簇的数目，其他一些算法选择合适的簇数目作为算法的一部分。如果算法需要指定簇的数目，那么就期望通过基于该值的实验能够取得好的结果

每个独立的算法也会有自己的一些特别之处，这些需要在算法评估时考虑，但是表6-2提供了一些总体上指导算法选择的方法。从这里开始，我们期望花一些时间对多个算法进行评估以确定哪种方法最适合于你的数据。

6.2.3 确定相似度

很多聚类算法都包含一个相似度的概念，该相似度用于确定一篇文档是否属于某个簇。在很多聚类算法中，相似度通过两篇文档的距离计算来实现。为了使这些距离计算方法有效，大部分系统都将文档表示成向量（基本总是稀疏向量，即大部分元素为零），其中向量中的每个元素是某个具体词项在某篇具体文档中的权重。该权重可以是应用所希望的任意值，但是通常都是TF-IDF的某个变形。之所以这些你听起来都依稀熟悉（实际上你也应该熟悉），是因为聚类中文档加权的方法与搜索中使用的方法类似。如果要重新复习一下这些概念，请参考3.2.3节。

实际上，文档向量几乎总是首先使用 p 范式（ $p \geq 0$ ）来进行归一化，这样，很短和很长的文档不会对结果造成负面影响。基于 p 范式归一化仅仅意味着将每个向量除

以它的长度，可以把所有向量缩放到单位形状上（例如，2范式就是一个单位圆）。最常用也是读者最熟悉的范式是1范式（曼哈顿距离）和2范式（欧氏距离）。你会注意到本章后面的例子中使用了欧氏距离来对向量做归一化处理。为了解更多的有关 p 范式的知识，请参考[http://en.wikipedia.org/wiki/Norm_\(mathematics\)](http://en.wikipedia.org/wiki/Norm_(mathematics))。

向量创建之后，将两个向量的距离看成两篇文档的距离就具有一定的合理性。存在很多不同的距离计算方法，下面只给出一些最常见的方法。

- 欧氏距离：这是一种久经考验并且效果很好的两个点之间的“箭头飞行”距离。一些变形包括平方欧氏距离（可以减少一次求平方根的过程）以及可以对向量部分元素加权的版本。
- 曼哈顿距离：由于该距离表示的是乘出租车沿网格线在城市（如纽约城的曼哈顿街区）内行进的距离，有时也称为出租车距离。有时，部分元素的计算可以加权。
- 余弦距离：向量尾部重合得到的夹角的余弦值，因此两篇相似文档（夹角为0）的余弦有可能为1。参考3.2.3节。

正如将要在有关Apache Mahout那一节看到的那样，距离计算方法往往可以作为参数输入，这样可以使用不同的距离进行试验。至于到底选择哪种距离计算方法，应该与应用于向量的归一化方法保持一致。比如，如果使用2范式，那么欧氏距离或者余弦距离可能最合适。话虽如此，尽管有些方法理论上不正确，但是却能在没有上述校准过程的情况下也有效。

对于概率方法而言，相似度问题实际上就是给定文档属于某个簇的概率问题。概率方法往往会有一个更复杂的模型来定义如何基于统计分布和其他性质来度量文档的相关性。此外，某些基于距离的方法（K-Means）也可以以概率的方式展示。

6.2.4 给聚类结果打标签

由于聚类往往用于真实用户的发现工具当中，因此选择好的标签和/或代表性文档往往与在聚类应用中确定簇本身一样重要。没有好的标签和代表性文档的话，用户会对与簇交互及寻找和发现有用文档望而却步。

从簇中选择代表性文档有多种做法。一种最基本的做法是，随机选择文档，给用户一个更宽泛的混合结果，有可能会带来新的发现，但是同时，如果文档远离中

心,那么就无法捕捉簇的主题内容。为纠正这一点,文档可以基于其离簇中心的邻近度或者到簇的隶属度来选择。这样的话,文档就可能很好地反映出簇的主题,但是此时也可能无法得到随机方法有时带来的意外发现。上述分析之后会得到一个双重策略,其中有些文档是随机选择的,而有些是基于邻近度/概率来选择的。

选择好的标签或主题则比选择代表性文档要难,目前存在很多做法,各有优缺点。某些应用中,类似多面展示(参考第3章)的简单技术可以用于高效展示出簇中的常见高频标签,但是大部分应用则需要寻找并展示簇中重要的词项和短语。当然,到底认定哪些重要是一个在研的研究主题。一个简单的方法是利用向量中的权重(比如我们熟悉的TF-IDF,参考3.2.3节)并按权重高低返回词项列表。使用 n 元组则可以对上述方法进行扩展,基于文档集/簇中的权重返回短语(严格地说它们是短语,但是它们的质量可能不高)列表。另一种普遍的做法是通过使用奇异值分解,比如LSI(参考Deerwester [1990])或LDA(参考Blei [2003],会在本章后面介绍)进行某种概念/主题建模。另一种有用的方法是使用簇内词项和簇外词项的对数似然率(log-likelihood ratio,简称LLR,参考Dunning [1993])。上述方法背后除TF-IDF(我们已经在前面讨论过)之外的数学原理超出了本书的讨论范围。但是在使用本章的工具(Carrot²和Apache Mahout)时,你会看到上述方法的展示,要么是以算法自身的一部分隐式展示,要么是以一个具体工具显式展示。不论如何获得簇的标签,最后的结果会对理解簇的质量有用,而这正是下一节的主题。

6.2.5 聚类结果的评估

利用任意文本处理工具,实验并评估聚类结果应该和设计架构或设想部署一样都是构建应用的重要一部分。与搜索、命名实体识别及本书中的其他概念一样,聚类可以采用多种方法来评估。

第一种被大部分人采用的方法称为严肃性测试,也称为嗅觉测试。做法是让对好结果已经有所认识的人去看看簇的结果,看看是否合理?尽管永远不会从嗅觉测试中读到太多的内容,它却是过程中不可缺少的一环,也是常常抓住输入参数错误或丢失这类愚蠢错误的手段。这种方法的缺点是不可能按需复制用户的反应,也不能重现,更不用说它只代表一个人的观点。不管怎样,它也依赖于标签的生成过程,而这个过程不一定能精确地捕捉簇的信息。

让一些人来标注结果往往是下一步要做的事情。不管是一个能保证质量的团队还是一组目标用户，对一组用户的反应进行关联分析可以提供非常有价值的反馈。但同时付出的代价是安排测试所花费的时间和金钱，还有人的错误以及无法按需重现等。但是如果做多次处理的话，可以导出黄金标准，下面接着介绍关于黄金标准的问题。

黄金标准是一个或多个用户创建的认为是聚类的理想结果的一系列簇集合。该集合一旦构建，就可以用于与多个聚类实验的结果进行比较。创建黄金标准往往不切实际、十分脆弱（处理更新、新文档之类的问题）或者对于大数据集来说过于昂贵。如果知道文档集变化不大并且有时间的話，那么构建黄金标准或者只对其中的子集构建黄金标准，有可能是值得的。一种半自动的方法是运行一次或多次聚类算法，然后让一个或多个用户手工通过调整簇来获得最后的结果。当判定结果就绪时，可以采用多种公式（纯度、归一化互信息、Rand指数、F值等）将聚类结果归结为单个表示聚类质量的指标。这里不给出这些公式，有兴趣的读者可以参考 *An Introduction to Information Retrieval* (Manning 2008) 的16.3节，那里给出了这些公式的具体计算方法。

最后，有些数学工具可以帮助评估聚类结果。这些工具都是启发式的聚类结果评估方法，不需要人工输入。它们不单独使用，而是作为聚类质量的辅助指标。第一种做法是随机去除输入数据中的某个子集然后运行聚类算法。聚类结束之后，计算每个簇内文档占总文档数目的比例并将它们放在一边。接下来，将随机数据放回，重新进行聚类，之后重新计算上述比例值。由于选出的留存数据是随机分布的，所以可以预期其分布与第一个集合中的分布基本一致。如果在第一个集合中，簇A占50%的文档比例，那么可以预期（并不能保证）在后一个更大集合中的比例也是50%。

一些来自信息论（参考http://en.wikipedia.org/wiki/Information_theory）的其他指标有助于评估聚类质量。第一个指标是熵。熵是度量随机变量不确定性的一项指标。实际上，它可以度量簇中包含的信息量。对于文本聚类而言，可以以熵为基础计算困惑度，后者度量的是簇隶属度预测所用词的好坏程度。

对于聚类还有很多可以学习的东西。对簇背后更多概念感兴趣的读者，*An Introduction to Information Retrieval* (Manning 2008) 这本书是一个好的起点。尤

其是该书的第16、17章更深入地集中关注这里介绍到的一些概念。Cutting等人在他们的论文“Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections”（Cutting 1992）中提供了如何利用聚类方法进行发现的方法。现在，我们将继续本章内容并考察多个聚类的实现，包括对搜索结果的聚类以及对文档集的聚类。

6.3 搭建一个简单的聚类应用

对于下面几节的讨论来说，我们会利用来自多个新闻网站RSS/Atom源的内容来展示聚类的概念。为此，我们已经构建了一个简单的Solr Home（schema、配置文件等），其处于solr-clustering目录，可以从多个新闻报纸和新闻机构获取数据源。该实例依赖Solr的Data Import Handler，可自动从数据源摄取并将信息索引到schema中。基于这些数据源，我们可以展示不同的聚类库，具体的介绍如下。

基于第3章获得的搜索知识，对于我们的新聚类应用而言，感兴趣的三个Solr片段就是shcema.xml、rss-data-config.xml和在solrconfig.xml中有关Data Import Handler的添加。对于schema和RSS配置而言，我们考察来自不同源的内容并将它们映射到一些公共字段中，然后进行索引。我们也存储词项向量，原因将在6.5.1节中给出。

有关Data Import Handler（DIH）配置的细节可以从Solr的维基百科页面（<http://wiki.apache.org/solr/DataImportHandler>）上找到。为在本书源码的聚类配置上运行Solr，需要在源码发布根目录下运行下列命令。

- `cd apache-solr/example`
- `./bin/start-solr.sh solr-clustering`
- 调用Data Import Handler导入命令：`http://localhost:8983/solr/dataimport?command=full-import`
- 检查导入的状态：`http://localhost:8983/solr/dataimport?command=status`

利用上述基本配置，现在就可以利用刚才提到的数据源索引结果在实战中展示聚类的过程。下面首先利用Carrot²对搜索结果聚类，然后考察利用Apache Mahout对文档集聚类。

6.4 利用Carrot²对搜索结果聚类

Carrot²是一款基于类BSD许可证发布的开源搜索结果聚类库，所在地址为：<http://project.carrot2.org/>。它特别设计为从典型的搜索结果（如标题和一小段文本）中产生高性能的结果。该软件库附带了对不同搜索API的支持，包括Google、Yahoo!、Lucene和Solr（作为客户端），并且能对XML或程序创建的文档聚类。此外，Solr项目已经将Carrot²集成到其服务器端，这一点我们后面会讨论。

Carrot²自带了两种聚类的实现，分别是STC（suffix tree clustering）和Lingo。

STC首先由Zamir和Etzioni引入到Web搜索结果聚类，具体参见论文“Web document clustering: a feasibility demonstration”（Zamir 1998）。算法基于后缀树数据结构，可以高效（线性时间）识别公共子串。快速寻找公共子串是快速寻找簇标签的关键之一。要阅读更多有关后缀树的知识，可以从http://en.wikipedia.org/wiki/Suffix_tree开始。

Lingo算法由Stanisław Osinski和Dawid Weiss（Carrot²项目的创建者）提出。Lingo使用了奇异值分解（SVD，参考http://en.wikipedia.org/wiki/Singular_value_decomposition）来寻找好的簇及短语来识别好的标签。

Carrot²也带了一个用户接口用于对自己的数据进行实验，以及一个支持REST的服务器端的实现，这样便于通过其他编程语言来与Carrot²进行交互。最后，如果聚类的结果太不均衡，应用可以通过一个良好定义的API把自己的聚类算法加入到框架中。要深入了解Carrot²，请参考<http://download.carrot2.org/head/manual/>上的手册。

本节的剩余部分将集中关注如何利用API来对数据源进行聚类，然后考察如何将Carrot²集成到Solr中。本节结束时考察算法的质量和速度性能。

6.4.1 使用Carrot²API

Carrot²的架构以流水线方式实现。从文档源获取内容之后，一个或多个模块依次对此进行修改并进行聚类，最后从另一端输出结果簇。就实际类而言，从最基本的层面上来说，整条流水线包含一个或多个受IController实现所控制的IProcessingComponents类。控制器对模块进行初始化，并以正确的次序和适当的输入来调用这些模块。IProcessingComponent实现的例子包括很多文档源（GoogleDocumentSource，YahooDocumentSource，LuceneDocumentSource），以及聚类

实现本身：STCClusteringAlgorithm和LingoClusteringAlgorithm。

很自然地，实现当中要使用的其他一些处理过程包括对文本的切词和词干还原处理。对于控制器而言，存在两种实现：一种是为简易设置和一次性使用而设计的SimpleController，而另一种是为生产环境中使用的CachingController。由于查询往往重复出现，因此可以对搜索结果进行缓存处理，CachingController 能够利用这一做法带来的好处。

为了解Carrot²的实际效果，下面看看用于对简单文档进行聚类的样本代码。第一步是创建一些文档。对于Carrot²而言，文档由三个元素构成：标题、摘要/片段和URL。给定这种性质的文档集合，可以按照如下清单所示，很直接地对它们进行聚类处理。

清单6-1 简单的Carrot²示例

```
//... setup some documents elsewhere
final Controller controller =
    ControllerFactory.createSimple ();           ← 构建 IController
documents = new ArrayList<Document> ();
for (int i = 0; i < titles.length; i++) {
    Document doc = new Document (titles[i], snippets[i],
        "file://foo_" + i + ".txt");
    documents.add (doc);
}
final ProcessingResult result = controller.process (documents,
    "red fox",
    LingoClusteringAlgorithm.class);           ← 对文档聚类
displayResults (result);                       ← 打印输出簇结果
```

运行上述代码产生的结果如下。

```
Cluster: Lamb
    Mary Loses Little Lamb. Wolf At Large.
    March Comes in like a Lamb
Cluster: Lazy Brown Dogs
    Red Fox jumps over Lazy Brown Dogs
    Lazy Brown Dogs Promise Revenge on Red Fox
```

尽管上述示例中的文档明显属于虚构（参考源码库中的Carrot2ExampleTest.java来了解文档的构造过程），上述代码还是能够很有效地展示Carrot² API使用

的简便性。很多应用不只是针对上述简单的情况，而是基于性能原因希望使用CachingController。正如该类的名字暗示的那样，CachingController会尽可能多地缓存结果以便提高性能。应用程序可能也希望使用其他的数据源（如Google或Yahoo!）或者实现自己的IDataSource来表示内容。此外，很多模块具有一系列属性，可以用来调节或修改结果的质量和响应速度，关于这一点将在6.7.2节中讨论。

刚才已经对利用Carrot²进行聚类的基本实现过程有所了解，下面我们看看如何将它与Solr集成。

6.4.2 使用Carrot²对Solr的搜索结果聚类

在1.4版中，Apache Solr增加了对基于Carrot²的搜索结果聚类的全面支持，包括通过solrconfig.xml文件、模块的所有属性以及用于聚类的算法的配置能力。很自然地，Carrot²对Solr的搜索结果来聚类，可以允许应用来定义到底哪些字段代表标题、摘要片段和URL。实际上，这些内容已经在本书附带源码库中的solr-clustering目录下搭建并配置好。

为在Solr中使用Carrot²聚类模块，需要做三部分的配置工作。首先，该模块实现为一个SearchComponent，这也意味着它可以插入到一个Solr RequestHandler中。配置该模块的XML文件如下所示：

```
<searchComponent
  class="org.apache.solr.handler.clustering.ClusteringComponent"
  name="cluster">
  <lst name="engine">
    <str name="name">default</str>
    <str name="carrot.algorithm"><lineArrow/>
      org.carrot2.clustering.lingo.LingoClusteringAlgorithm</str>
  </lst>
  <lst name="engine">
    <str name="name">stc</str>
    <str name="carrot.algorithm"><lineArrow/>
      org.carrot2.clustering.stc.STCClusteringAlgorithm</str>
  </lst>
</searchComponent>
```

在<searchComponent>声明中，建立了ClusteringComponent然后给出所使用的

Carrot²聚类算法。本例当中，我们同时构建了Lingo和STC聚类算法。下一步将SearchComponent挂钩到一个RequestHandler中，如：

```
<requestHandler name="standard"
  class="solr.StandardRequestHandler" default="true">
  <!-- default values for query parameters -->
  <!-- ... -->
  <arr name="last-components">
    <str>cluster</str>
  </arr>
</requestHandler>
```

最后，在Solr中往往会建立一些智能的默认值，来避免各种参数必须通过命令行来传递。在本例中，我们使用的默认值如下：

```
<requestHandler name="standard"
  class="solr.StandardRequestHandler" default="true">
  <!-- default values for query parameters -->
  <lst name="defaults">
    <!-- ... -->
    <!-- Clustering -->
    <!--<bool name="clustering">true</bool>-->
    <str name="clustering.engine">default</str>
    <bool name="clustering.results">true</bool>
    <!-- The title field -->
    <str name="carrot.title">title</str>
    <!-- The field to cluster on -->
    <str name="carrot.snippet">desc</str>
    <str name="carrot.url">link</str>
    <!-- produce summaries -->
    <bool name="carrot.produceSummary">>false</bool>
    <!-- produce sub clusters -->
    <bool name="carrot.outputSubClusters">>false</bool>
  </lst>
</requestHandler>
```

在默认参数的配置当中，我们声明Solr使用默认聚类引擎（Lingo），Carrot²使用Solr标题字段作为其标题，Solr描述字段作为其文本片段字段、Solr链接字段作为Carrot² URL字段。最后一点，我们通知Carrot²产生摘要但是不输出子簇（为准确起见，这里的子簇指的是在单个簇内聚类的输出结果）。

这就是搭建过程所有需要做的事情！假定Solr按照清单6-3所示启动，那么要求Solr返回搜索结果簇就十分容易，只需要将参数`&clustering=true`加到URL中即可，就像`http://localhost:8983/solr/select/?q=*&clustering=true&rows=100`中的一样。执行这条命令后，Solr会从索引中返回100篇文档并对它们进行聚类。上述聚类查询的运行结果的一个截屏如图6-2所示。

```

- <lst>
  - <arr name="labels">
    <str>Overtime</str>
    <str>Minnesota Vikings</str>
    <str>Bears Beat Vikings</str>
  </arr>
  + <arr name="docs"></arr>
</lst>
- <lst>
  - <arr name="labels">
    <str>Texas Tech Suspends</str>
    <str>Player after a Concussion</str>
    <str>Tech Suspended Mike Leach</str>
  </arr>
  - <arr name="docs">
    - <str>
      http://www.nytimes.com/aponline/2009/12/28/sports/AP-FBC-T25-Texas-Tech-Leach-Suspended.html
    </str>
    <str>761b5a908469a491fb58782175c4b19b</str>
    <str>a5a74692f6bd858a630324aebccc47de</str>
  </arr>
</lst>
- <lst>
  - <arr name="labels">
    <str>PORTLAND</str>
    <str>Sixers</str>
    <str>Trail Blazers</str>
  </arr>
  - <arr name="docs">
    <str>00493468085a22165e409053d3b2c87f</str>
    <str>439a216eb8832b259b8203318b623d33</str>
    <str>6c677f6d6cd2fa744c76cb12daad1723</str>
    <str>d7c18a049c230eeb428c99f19699fd5a</str>
    <str>0dfd31d031f5eba2f6153acc9afee5d1</str>
  </arr>
</lst>
- <lst>
  - <arr name="labels">
    <str>R Reuters sportsNews 4</str>

```

图6-2 运行Solr聚类命令后的一个截屏图

在图6-2的底部，我们有意地放了一条垃圾结果*R Reuters sportsNews 4*，这是为了展示必须要通过多个属性对Carrot²进行适当调节来获得更好的结果，这一点会在6.7.2

中讨论。为完整了解Solr中聚类模块可以调节的选项，请参考<http://wiki.apache.org/solr/ClusteringComponent>。

现在已经对搜索结果的聚类有所了解，下面介绍如何利用Apache Mahout来对整个文档集聚类。在后面考察性能时我们还会回到Carrot²。

6.5 利用Apache Mahout对文档集聚类

Apache Mahout是Apache软件基金会的一个项目，其目标是开发一套机器学习库，该库设计成能够支持基本的处理，也能扩展到大量输入的情况。在本书写作时，它包含了分类、聚类、协同推荐、进化编程及更多算法，也包含了解决机器学习问题的有用工具，如矩阵处理及Java原生类型（存储整型、双浮点数等类型的Map、List和Set）的存储工具。很多情况下，Mahout依赖于Apache Hadoop（<http://hadoop.apache.org>）框架，（通过MapReduce编程模型和一个分布式文件系统HDFS）服务于开发可扩展的算法。尽管本章大部分内容集中关注基于Mahout的聚类，第7章也会介绍基于Mahout的分类。Mahout的其他部分内容可以从其网站<http://mahout.apache.org/>和*Mahout in Action*一书（参考<http://manning.com/owen/>）中查到。为开始本节内容，需要从<http://archive.apache.org/dist/mahout/0.6/mahout-distribution-0.6.tar.gz>下载Mahout 0.6版并将它解压到一个目录下，从现在开始我们就把这个目录称为\$MAHOUT_HOME。下载并解压之后，转到\$MAHOUT_HOME目录并运行`mvn install -DskipTests`（你可以运行测试，但是需要花费很长时间！）。

为了避免麻烦，接下来的三节会考察如何准备数据，然后利用Apache Mahout中的K-means的实现来进行聚类。

Apache Hadoop——拥有极大计算能力的黄色小象

Hadoop是Google提出的思想（参考Dean [2004]）的一个实现，首先在Lucene项目Nutch中实现，后来成为Apache软件基金会自己的项目。其基本思想是一个分布式文件系统（Google称为GFS，Hadoop称为HDFS）搭配一个编程模型（MapReduce），这样几乎没有任何并行和分布式系统背景的工程师，也能编写出既有扩展性又有容错性的能在极大规模计算机集群上运行的程序。

尽管并非所有的应用都能采用MapReduce模型来编写，很多基于文本的应用

非常适合采用这个方法。

有关Apache Hadoop的更多信息，请参考 [Tom Whiter的*Hadoop: The Definitive Guide*] (<http://oreilly.com/catalog/9780596521981>) 和Chuck Lam的 [*Hadoop in Action*] (<http://manning.com/lam/>) 。

6.5.1 对聚类的数据进行预处理

Mahout聚类要求数据采用org.apache.mahout.matrix.Vector格式。Mahout中的Vector就是一个简单的浮点元组，比如<0.5, 1.9, 100.5>。更一般地说，向量，通常也称为特征向量，是机器学习中常用的一种数据结构，用以表示系统中文档或其他数据片的特性。向量通常要不很稠密要不很稀疏，这取决于数据本身。对于文本应用来说，这里的向量往往是稀疏的，这是因为在所有文档集中的词项很多，但是在任一具体文档中的词项又很少。幸运的是，在常见的机器学习任务的计算中，稀疏性往往会体现出优点。自然而然，Mahout通过继承Vector实现了多种向量类型，以表示稀疏向量和稠密向量。这些实现的名称分别是org.apache.mahout.matrix.SparseVector和org.apache.mahout.matrix.DenseVector。当运行自己的应用时，你需要对数据取样以确定数据是否稀疏，然后选择合适的表示。你可以在自己的数据子集上尝试两种表示方法以确定哪种更好。

Mahout中有多种方法可以为聚类创建Vector。

- 编程法：写一段代码对Vector实例化然后将它保存到合适的位置。
- Apache Lucene索引：将Apache Lucene的索引转换为Vector集合。
- Weka的ARFF格式：Weka是新西兰怀卡托大学的一个机器学习项目，该项目中定义了一种ARFF格式。可以从<http://cwiki.apache.org/MAHOUT/creating-vectors-from-wekas-arff-format.html>获得更多有关ARFF格式的信息。而有关Weka的信息，请参考Witten和Frank写的 *Mining: Practical Machine Learning Tools and Techniques (Third Edition)* 一书 (<http://www.cs.waikato.ac.nz/~ml/weka/book.html>) 。

由于本书中不使用Weka，所以下面就不讨论ARFF格式，而直接关注Mahout中前两种产生Vector的方法。

通过编程创建向量

通过编程创建Vector十分直接,如清单6-2所示。

清单6-2 利用Mahout创建Vector

标签为
parse、
为3000的
seVector

稀疏向量
前三个元
赋值

稠密向量
向量的
不同,因
为不相等

```
double[] vals = new double[]{0.3, 1.8, 200.228};
```

```
Vector dense = new DenseVector (vals);
```

```
assertTrue (dense.size () == 3);
```

```
Vector sparseSame = new SequentialAccessSparseVector (3);
```

```
Vector sparse = new SequentialAccessSparseVector (3000);
```

```
for (int i = 0; i < vals.length; i++) {
```

```
    sparseSame.set (i, vals[i]);
```

```
    sparse.set (i, vals[i]);
```

```
}
```

```
assertFalse (dense.equals (sparse));
```

```
assertEquals (dense, sparseSame);
```

```
assertFalse (sparse.equals (sparseSame));
```

构建标签为 my-dense 并带有 3 个值的 DenseVector, 该向量的维度为 3。

构建标签为 my-sparse-same、维度为 3 的向量

由于稠密向量和稀疏向量的值和维度都一样, 因此它们相等

当从数据库或者其他Mahout不支持的数据源中读取数据时,通常采用编程方式来构建向量。当构建Vector之后,必须要写入一个Mahout理解的格式。所有Mahout中的聚类算法预期的都是一个或多个Hadoop SequenceFile格式的文件。Mahout中提供了org.apache.mahout.utils.vectors.io.SequenceFileVectorWriter类,辅助将Vector序列化成正当格式。清单6-3给出了这一过程。

清单6-3 将向量序列化成SequenceFile

构建 Hadoop
SequenceFile,
Writer 负责完成
到一个 HDFS
文件的物理写入

VectorWriter
量进行处
调用下面
SequenceFile.
的写入方
法

```
File tmpDir = new File (System.getProperty ("java.io.tmpdir"));
```

```
File tmpLoc = new File (tmpDir, "sfvwt");
```

```
tmpLoc.mkdirs ();
```

```
File tmpFile = File.createTempFile ("sfvwt", ".dat", tmpLoc);
```

```
Path path = new Path (tmpFile.getAbsolutePath ());
```

```
Configuration conf = new Configuration ();
```

```
FileSystem fs = FileSystem.get (conf);
```

```
SequenceFile.Writer seqWriter = SequenceFile.createWriter (fs, conf,
```

```
    path, LongWritable.class, VectorWritable.class);
```

```
VectorWriter vecWriter = new SequenceFileVectorWriter (seqWriter);
```

```
List<Vector> vectors = new ArrayList<Vector> ();
```

```
vectors.add (sparse);
```

```
vectors.add (sparseSame);
```

```
vecWriter.write (vectors);
```

```
vecWriter.close ();
```

为 Hadoop 构建配置

执行文件的写入任务

Mahout也能将Vector写成JSON格式，但是这样做纯粹是为了满足人们阅读的需要，因为在运行时对它们进行序列化和反序列化的处理很慢，会显著降低聚类算法的速度。由于我们使用的是Solr，其背后是Apache Lucene，因此下一节基于Lucene的索引来构建向量要有趣得多。

基于Apache Lucene索引构建向量

假设为Vector构建所用的字段是基于schema中设定的termVector="true"的选项来建立的，那么一个或多个Lucene索引是构建Vector的很好资源。上述字段的构建示例如下：

```
<field name="description" type="text"
      indexed="true" stored="true"
      termVector="true"/>
```

给定一个索引，可以使用Mahout Lucene的工具来将索引转换成包含Vector的SequenceFile。转换过程可以在命令行通过运行org.apache.mahout.utils.vector.lucene.Driver程序来完成。尽管Driver程序有很多选项，表6-3列出了其中一些更常用的选项。

表 6-3 Lucene 索引转换选项

参 数	描 述	是 否 必 需
--dir <Path>	指定 Lucene 索引的位置	Yes
--output <Path>	将 SequenceFile 输出到文件系统的路径	Yes
--field <String>	作为源使用的 Lucene 字段的名字	Yes
--idField <String>	包含文档唯一 ID 的 Lucene 字段的名字，可以用于给向量赋予标签	No
--weight [tf tfidf] The type	字段中词项权重的类型。TF 表示只使用词项频率，而 TF-IDF 表示同时使用词项频率和逆文档频率	No
--dictOut <Path>	词项及其在向量中位置这种映射信息输出的位置	Yes
--norm [INF -1 A double >= 0]	给出向量归一化的方法。参考 http://en.wikipedia.org/wiki/Lp_norm .	No

为在我们的Solr实例中实现上述转换，我们可以点击包含Lucene索引的目录下的driver，并指定合适的输入参数，而driver就会完成剩余的工作。为了展示，我们假定

Solr的索引存储在<Solr Home>/data/index下,索引已经像本章前面介绍的那样完成了构建过程。你可以通过如清单6-4所示的那样,运行driver来生成Vector。

清单6-4 从Lucene索引转换为Vector的样本代码

```
<MAHOUT_HOME>/bin/mahout lucene.vector
--dir <PATH>/solr-clustering/data/index
--output /tmp/solr-clust-n2/part-out.vec --field description
--idField id --dictOut /tmp/solr-clust-n2/dictionary.txt --norm 2
```

在清单6-4的例子中, driver程序接受Lucene索引,从索引中获取必要的文档信息,然后将它写到part-out.dat文件中(在Mahout/Hadoop中, *part*很重要)。已构建的dictionary.txt文件将包含索引中词项到创建的向量中的位置映射。这对于后面出于显示目的重建向量十分重要。

最后,我们选择的是2范式归一化,这样我们就可以利用Mahout中的CosineDistanceMeasure进行聚类。现在我们拥有了一些向量,下面利用Mahout中的K-means实现来进行聚类。

6.5.2 K-means聚类

不论是在广阔的机器学习社区还是Mahout内部,都有多种聚类算法。例如,在本书写作期间, Mahout自己就实现了如下聚类算法:

- Canopy
- Mean-Shift
- Dirichlet
- Spectral
- K-Means和模糊K-Means

在这些选择当中, K-Means是最出名的一个。K-Means是一个简单直观的聚类算法,往往能够很快地生成很好的结果。它根据距离计算结果以反复迭代的方式不断地将文档加入到 k 个簇中的一个来完成聚类,该距离计算的是文档和簇中心之间的距离,由用户提供的某个距离计算方法来确定。每次迭代最后,中心会重新计算。整个过程在聚类结果基本没有什么修改或者达到最大迭代次数之后停止。聚类过程要么以某些初始中心为种子开始,要么通过从输入数据集的向量集合当中随机选择中

心开始。K-Means确实有一些缺点。首先，必须要选择 k ，而不同的 k 很自然会得到不同的结果。此外，初始中心的选择对结果的影响很大，因此一定要保证多次尝试中采用不同的值。最后，和大部分技术一样，一种明智的做法是，利用不同参数进行多次迭代以确定哪种参数与你的数据最吻合。

在Mahout中运行K-Means聚类算法就像以合适参数运行org.apache.mahout.clustering.kmeans.KMeansDriver类一样简单。借助于Hadoop的强大威力，聚类算法可以在单机模式或分布式模式（在Hadoop集群）下运行。就本书的目的而言，我们将使用单机模式，但是使用分布式模式也没有太大不同。

在考察KMeansDriver的不同选项之前，我们直接进入一个对前面构建的Vector转存结果进行聚类的例子。清单6-5给出了利用命令行运行KMeansDriver的示例。

清单6-5 使用KMeansDriver命令行工具的例子

```
<$MAHOUT_HOME>/bin/mahout kmeans \  
  --input /tmp/solr-clust-n2/part-out.vec \  
  --clusters /tmp/solr-clust-n2/out/clusters -k 10 \  
  --output /tmp/solr-clust-n2/out/ --distanceMeasure \  
  org.apache.mahout.common.distance.CosineDistanceMeasure \  
  --convergenceDelta 0.001 --overwrite --maxIter 50 -clustering
```

大部分参数都可以从名字本身去理解，因此我们集中关注K-Means算法中的6个主要输入。

- --k: K-Means中的 k 。指定了返回簇的个数。
- --distanceMeasure: 指定用于计算文档到中心的距离计算方法。本例中我们使用的是余弦相似度计算方法（如果记得的话，这与Lucene/Solr的运行类似）。在org.apache.mahout.common.distance包中Mahout自带了多种距离计算方法。
- --convergenceDelta: 定义了一个阈值，低于该阈值则认为聚类结果收敛，算法可以结束。默认值为0.5，这里取0.001完全是随意的。用户可以用该值进行实验来实现时间和质量之间的折中。
- --clusters: 该路径包含了一开始聚类的种子中心。如果--k没有显式指定，该路径就必须包含一个，个向量的文件（按照清单6-3的方式进行序列化处理）。如果--k指定的话，那么就从输入当中随机选择 k 个向量。

- `--maxIter`: 指定了一个最大的迭代次数, 如果在此之前算法没有收敛, 则到达该次数之后退出迭代。
- `--clustering`: 利用额外时间输出每个簇的成员。如果该选项关闭, 则只需要确定簇的中心。

当运行清单6-5的命令时, 你会看到一系列登录的信息不断闪过, 理想情况下没有任何错误和异常。一旦结束之后, 输出目录下应该包含多个子目录、输入簇(本例中是随机生成的簇)以及点到最终迭代输出簇的映射信息, 每个子目录下包含每次迭代的输出(子目录的名称为clusters-X, 其中X是迭代次数)。

由于Hadoop的序列文件自身是输出结果, 因此它们的原始形式对人不可读。但是Mahout带了多个可用于浏览聚类结果的工具。其中最有用的一个是org.apache.mahout.utils.clustering.ClusterDumper, 但是org.apache.mahout.utils.ClusterLabels, org.apache.mahout.utils.SequenceFileDumper和 org.apache.mahout.utils.vectors.VectorDumper也很有用。这里我们将集中关注ClusterDumper。正如可以从名称猜到的那样, ClusterDumper设计为将簇转存到控制台窗口或者具有可读格式的文件中。例如, 为浏览前面给出的运行KMeansDriver命令的结果, 尝试一下如下的命令:

```
<MAHOUT_HOME>/bin/mahout clusterdump \
--seqFileDir /tmp/solr-clust-n2/out/clusters-2 \
--dictionary /tmp/solr-clust-n2/dictionary.txt --substring 100 \
--pointsDir /tmp/solr-clust-n2/out/points/
```

在这个典型实例当中, 我们通知程序包含簇(`--seqFileDir`)、词典(`--dictionary`)和原始数据点(`--pointsDir`)的目录, 同时也通知程序在打印簇向量中心时将其截短, 只输出100个字符(`--substring`), 这样结果就更清晰。上述实例运行在2010年7月5日新闻所建立的索引上, 会产生如下结果。

```
:C-129069: [0:0.002, 00:0.000, 000:0.002, 001:0.000, 0011:0.000, \
002:0.000, 0022:0.000, 003:0.000, 00
Top Terms:
    time                =>0.027667414950403202
    a                   => 0.02749764550779345
    second              => 0.01952658941437323
    cup                 =>0.018764212101531803
    world               =>0.018431212697043415
```

```

won =>0.017260178342226474
his => 0.01582891691616071
team =>0.015548434499094444
first =>0.014986381107308856
final =>0.014441638909228182

:C-129183: [0:0.001, 00:0.000, 000:0.003, 00000000235:0.000, \
001:0.000, 002:0.000, 01:0.000, 010:0.00
Top Terms:
a => 0.05480601091954865
year =>0.029166628670521253
after =>0.027443270009727756
his =>0.027223628226736487
polic => 0.02445617250281346
he =>0.023918227316575336
old => 0.02345876269515748
yearold =>0.020744182153039508
man =>0.018830109266458044
said =>0.018101838778995336
...

```

在上述输出示例中，ClusterDumper输出簇中心向量的ID，同时基于词项频率给出一些簇中的常见词项。近距离考察排名靠前的词项会发现，尽管有很多好的词项，也存在很多很差的词项，比如有一些停用词（a、his、said等）。这里暂时不考虑这个问题，在6.7节中我们还会谈到这个问题。

尽管简单地输出簇结果往往十分有用，很多应用需要能够概括内容的简洁标签，这一点在前面6.2.4节中有所讨论。Mahout中的ClusterLabels类是一个从Lucene（Solr）索引中生成标签的工具，能够用于提供很好描述簇内容的若干词。为在我们前面的聚类结果中运行ClusterLabels程序，在命令行运行如下命令，运行的目录和其他命令一样。

```

<MAHOUT_HOME>/bin/mahout \
org.apache.mahout.utils.vectors.lucene.ClusterLabels \
--dir /Volumes/Content/grantingersoll/data/solr-clustering/data/index/\
--field desc-clustering --idField id \
--seqFileDir /tmp/solr-clust-n2/out/clusters-2 \
--pointsDir /tmp/solr-clust-n2/out/clusteredPoints/ \
--minClusterSize 5 --maxLabels 10

```

在本例中，我们通知程序的信息当中有很多与前面从索引中抽取内容时一

样，比如索引的位置和所用的字段。我们也会加入簇和原始点所在位置的信息。`minClusterSize`参数设立一个阈值，该阈值指定了为计算标签簇当中最少的文档数目。这在对确实很大的文档集进行聚类且簇也很大时能够派上用场，因为此时应用可能希望将那些较小的簇看成离群点而忽略。`maxLabels`参数给出的是簇中获取的最大的标签数目。在本章前面构建的样例数据上运行上述命令的结果如下（为简洁起见做了缩短处理）。

Top labels for Cluster 129069 containing 15306 vectors

Term	LLR	In-ClusterDF	Out-ClusterDF
team	8060.366745727311	3611	2768
cup	6755.711004478377	2193	645
world	4056.4488459853746	2323	2553
reuter	3615.368447394372	1589	1058
season	3225.423844734556	2112	2768
olymp	2999.597569386533	1382	1004
championship	1953.5632186210423	963	781
player	1881.6121935029223	1289	1735
coach	1868.9364836380992	1441	2238
u	1545.0658127206843	35	7101

Top labels for Cluster 129183 containing 12789 vectors

Term	LLR	In-ClusterDF	Out-ClusterDF
polic	13440.84178933248	3379	550
yearold	9383.680822917435	2435	427
old	8992.130047334154	2798	1145
man	6717.213290851054	2315	1251
kill	5406.968016825078	1921	1098
year	4424.897345832258	4020	10379
charg	3423.4684087312926	1479	1289
arrest	2924.1845144664694	1015	512
murder	2706.5352747719735	735	138
death	2507.451017449319	1016	755

...

在输出结果中，每一列分别是

- 词项 (*Term*)：标签。
- *LLR* (对数似然率)：LLR基于Lucene索引中的多种统计量计算词项的好坏程度。为了解更多LLR的信息，参考http://en.wikipedia.org/wiki/Likelihood-ratio_test。

- *In-ClusterDF*: 词项在簇内出现的文档数目。该值和*Out-ClusterDF*用于计算LLR。

- *Out-ClusterDF*: 词项在某个簇外出现的文档数目。

在前面ClusterDumper给出的排名最高词项的例子中，近距离考察会发现一些好的词项（不考虑它们进行了词干还原处理）和一些几乎没用的词项。需要指出的是，大部分词项都能很好地描述簇中文档的主要内容。正如前面提到的那样，我们会在6.7节中考察如何对此进行改进。现在，我们看看如何利用Mahout的聚类功能，通过文档中的词距离识别其主题。

6.6 利用Apache Mahout进行主题建模

Mahout不仅有对文档聚类的工具，也有主题建模的实现，而主题建模在应用到文本时可以看成是词级别的聚类。Mahout中的唯一主题建模实现是LDA算法。而LDA（Deerwester 1990）是一种

... 面向离散型数据（如文本语料）集合的生成式概率模型。LDA是一个三层的贝叶斯模型，其中数据集合中的每个元素建模为下面的主题集合上的一个有限混合分布。而每个主题建模为下面的主题概率集合上的有限混合分布。

用非专业人员的话来说，LDA是一个将词簇转变为主题的算法，其基于假设文档与多个不同的主题有关，但是不确定到底哪篇文档与哪个主题相关，不确定的还有每个主题的确切标签。尽管从表面上看起来LDA没什么用，将主题词与文档集关联起来却具有一定价值。例如，可以在Solr中使用这些信息以在搜索应用中构建更多的发现功能。或者，它们也可以简明扼要地概括大型文档集的内容。主题词项也可以用于其他任务，比如分类和协同过滤（为了解更多有关这些应用的信息，请参考Deerwester [1990]）。现在，我们看看如何运行Mahout中的LDA实现。

为在Mahout中开始使用LDA，需要一些向量。正如前面清单6-3之后给出的那样，可以从Lucene索引中构建向量。但是对于LDA需要做一点小小的改动，这里只使用TF而不是默认的TF-IDF来计算权重，这是由于算法计算其内部统计量的方式而导致的结果。这种做法如下所示。

```
<MAHOUT_HOME>/bin/mahout lucene.vector \  
--dir <PATH TO INDEX>/solr-clustering/data/index/ \  
--output /tmp/lda-solr-clust/part-out.vec \  
--field desc-clustering --idField id \  
--dictOut /tmp/lda-solr-clust/dictionary.txt \  
--norm 2 --weight TF
```

该例几乎与前面的例子一模一样，只是输出的路径和`--weight`输入参数的不同，这里使用的是`--weight`参数是TF。

有了向量之后，下一步就是运行LDA算法，如下所示。

```
<MAHOUT_HOME>/bin/mahout lda --input /tmp/lda-solr-clust/part-out.vec \  
--output /tmp/lda-solr-clust/output --numTopics 30 --numWords 61812
```

尽管大部分参数的含义都很明显，这里还是有些事情值得解释一下。首先，我们给该应用赋予了额外内存。LDA是一个相当消耗内存的应用，因此必须要给它更多内存。其次，我们要求 `LDADriver` 从向量中识别30个主题（`--numTopics`）。与 `K-Means` 中指定簇的个数类似，LDA需要指定主题的个数。不管好坏，这意味着在你的应用中为了确定合适的主题数目需要反复试错。在看了参数分别为10和20的结果之后，我们会发现设置成30相当不科学。最后，`--numWords` 参数是所有向量中的词个数。当使用这里给出的向量创建方法时，词的数目可以很容易地从 `dictionary.txt` 文件的第一行获得。运行LDA之后，输出目录会包含一系列名称为 `state-*` 的目录，比如 `state-1`、`state-2` 等。目录的数目取决于输入和其他参数。最大数目对应的目录代表了最后的结果。

很自然地，运行LDA之后，你想看看结果如何。默认情况下LDA并不打印出这些结果。但是Mahout自带了一个简便工具来打印出主题，其名称为 `LDAPrintTopics`。其需要三个必需的输入参数和一个可选参数。

`--input`: 包含LDA运行输出结果的状态目录。这可以是任意状态目录而不一定是最后创建的那个目录。该参数为必需参数。

`--output`: 结果写入的目录。必需参数。

`--dict`: 用于创建向量的词项词典。必需参数。

`--words`: 每个主题需要打印的词数目。可选参数。

对于前面的LDA运行例子，`LDAPrintTopics` 可以按照如下方式运行：

```
java -cp "*" \  
    org.apache.mahout.clustering.lda.LDAPrintTopics \  
    --input ./lda-solr-clust/output/state-118/ \  
    --output lda-solr-clust/topics \  
    --dict lda-solr-clust/dictionary.txt --words 20
```

本例当中，我们想要state-118目录（正好是最后一个目录）下的排名最高的20个词。运行该命令会在主题输出目录下生成30个文件，每个文件代表一个主题。比如，主题22如下所示：

```
Topic 22  
=====  
yearold  
old  
cowboy  
texas  
14  
second  
year  
manag  
3414  
quarter  
opera  
girl  
philadelphia  
eagl  
arlington  
which  
dalla  
34  
counti  
five  
differ  
1996  
tri  
wide  
toni  
regul  
straight  
stadium  
romo  
twitter
```

观察一下这个类别中排名靠前的词，该主题可能是有关Dallas Cowboys在NFL季后赛中击败Philadelphia Eagles的事情，我们运行例子的时间正好是那场比赛结束的第二天。而且，尽管有些词看上去像离群点（如opera、girl），但是从大部分词语中可以获得主题的意义。在索引中搜索上述某些词项会发现，实际上确实有文章正好与这个事件有关，包括一篇有关Eagles接球员DeSean Jackson预测Eagles将会击败Cowboys（这些运动员就无法学乖吗？）的一则推文。这就是在Apache Mahout中运行LDA所需要知道的全部知识。接下来，我们考察Carrot²和Mahout的聚类性能。

6.7 考察聚类性能

像任一现实世界的应用一样，当程序员已经对如何运行应用有基本了解时，他们的心思很快就变成如何将该应用投入生产环境。为回答这个问题，需要从定性和定量两个方面来度量性能。我们首先考察提高质量的特征选择和约简方法，然后考察算法选择以及Carrot²和Apache Mahout的输入参数。最后通过在Amazon上的一个称为EC的按需计算功能来做一些基准测试。而对于Amazon基准测试，我们要感谢Timothy Potter和Szymon Chojnacki两人，在测试中我们会处理一个大型的邮件内容语料以了解Mahout的跨多机的性能。

6.7.1 特征选择与特征约简

特征选择与特征约简技术的设计目标是提高结果质量或者减少处理的内容。特征选择关注于预先选择好的特征，该过程可以是预处理的一部分也可能是算法输入的一部分。而特征约简关注那些价值很小的特征的去除，它是一个自动过程的一部分。两种技术往往都十分有益，部分原因如下所示：

- 减小问题的规模会使问题在计算和存储开销的意义上可解。
- 通过去掉数据中的一些噪声（如停用词）来提高质量。
- 有助于可视化和后处理，太多特征会阻塞用户界面和后续处理技术。

从很多方面而言，感谢第3章的工作，使得你对特征约简已经有所了解。在那一章中，我们使用了多种分析技术来减少搜索的词项数目，比如去除停用词和词干还原。这些技术也有助于聚类结果的提高。此外，在聚类中，甚至更激进的特征选择

往往会效果更好，由于处理的文档规模往往很大，因此预先约简特征会节省大量时间。

例如，对于本章的例子而言，我们使用了一个与前面搜索中不同的停用词文件，其区别在于，聚类中的停用词（考察源码库中的stopwords-clustering.txt文件）是先前所使用集合的超集。为构建stopwords-clustering.txt文件，我们考察索引中出现频率最高的那些词项并且运行多次聚类迭代过程来确定哪些词应该去掉。

不幸的是，这种方法本质上是临时性的，需要执行大量工作。并且该方法无法在语言之间移植，甚至在不同语料库之间也无法移植。为使得该方法更具移植性，应用程序通常试图根据词项权重（使用TF-IDF或其他方法）来去除词项，然后反复观察某个结果指标来确定聚类效果是否提高。例如，参考文献部分（Dash [2000], Dash [2002], and Liu [2003]）给出了很多方法及其讨论。也可以使用Mahout中已经集成的奇异值分解（SVD）方法来显著降低输入的规模。也要注意Carrot²的Lingo算法建立在SVD之上，因此对于Carrot²无须做任何事情。

奇异值分解是一种一般性的特征约简技术（这意味着它不仅限于聚类），它通过保留那些“重要”的特征并去掉不重要的特征来对原始数据集进行降维处理（通常在文本聚类中，每个独立的词代表 n 维向量的一个元素）。该降维过程是一个有损过程，因此不是没有风险，但是通常来说它能显著节省存储和CPU的开销。用重要性的概念来说的话，上述算法常常可以比作在语料库中抽取概念，但是并不能保证一定能获得这样的结果。对数学爱好者来说，SVD是矩阵（我们将用于聚类的文档表示成矩阵）因子分解成特征向量和其他部分的过程。这里我们把数学推导的细节留给其他人，读者可以参考<https://cwiki.apache.org/confluence/display/MAHOUT/Dimensional+Reduction>来获得更多Mahout中的SVD实现，以及多个SVD入门讲座和解释的链接。

要开始使用Mahout的奇异值分解，我们再次依赖bin/mahout这个命令行工具来运行算法。在Mahout运行SVD需要两步。第一步对矩阵进行分解，第二步进行某些清洗计算工作。在前面构建的聚类矩阵上运行SVD的第一步看起来像下面这样。

```
<MAHOUT_HOME>/bin/mahout svd --input /tmp/solr-clust-n2/part-out.vec \  
--tempDir /tmp/solr-clust-n2/svdTemp \  

```



```
--output /tmp/solr-clust-n2/svdOut \  
--rank 200 --numCols 65458 --numRows 130103
```

在本例中，我们具有通常的输入类型，比如输入向量的位置（`--Dmapred.input.dir`）及系统使用的一个临时位置（`--tempDir`），以及一些SVD特定的选项。

- `--rank`：指定输出矩阵的秩。
- `--numCols`：矩阵中的总列数。本例中为语料库中独立词项的数目，该值可以从/tmp/solr-clust-n2/dictionary.txt文件的开始部分找到。

在上述选项中，`rank`用于确定结果的输出样式，也是最难选择最优值的一个选项。通常来说需要从较小值（比如50）开始不断增加的反复试错过程。按照Mahout提交者和Mahout SVD代码的原作者Jake Mannix（Mannix 2010, July）的说法，对于文本问题，200到400之间的值比较好。很显然，需要进行多次运行尝试才能确定哪个`rank`值能够产生最好的结果。

在主SVD算法运行之后，必须要执行某单个清洗任务来产生最后的结果，如下所示。

```
<MAHOUT_HOME>/bin/mahout cleansvd \  
--eigenInput /tmp/solr-clust-n2/svdOut \  
--corpusInput /tmp/solr-clust-n2/part-out.vec \  
--output /tmp/solr-clust-n2/svdFinal --maxError 0.1 \  
--minEigenvalue 10.0
```

这一步的关键是选择误差阈值（`--maxError`）和最小的特征值（`--minEigenvalue`）。对于最小特征值来说，选0总是安全的，但是你可能希望选择一个更大的值。对于最大误差值，反复试错及使用聚类算法的输出会更好地理解性能（阅读Mannix [2010, August]来获得更多细节）。

正如你看到的那样，有很多方法可以用于选择特征或降低问题的规模。和大部分这种性质的事情一样，必须要实验来确定哪种选择与你的情况最吻合。最后，如果在聚类中使用SVD的结果（这是本节的要点，对吗？），那么还剩下最后一步。需要将原始矩阵的转置乘上SVD输出结果的转置。这些工作都可以使用`bin/mahout`命令来实现，只要输入正确的参数即可。有关这项验证留作读者的习题。接下来，我们要考察Carrot²和Apache Mahout的定量性能。

6.7.2 Carrot²的性能和质量

当涉及Carrot²结果的性能和质量时，它提供大量可供调节的参数，这还不包括首先选择的多种算法。下面会对算法的性能进行简单的考察，而有关所有参数现象的深入讨论则请参考Carrot²的手册。

选择Carrot²算法

首先，STC和Lingo算法有一个共同点，即在这两个算法中文档可能属于一个或多个簇。除此之外，两个算法为获取结果而采用的方法也不同。通常来说，Lingo产生的标签会比STC更好，但是付出的代价就是更慢，这点可以从图6-3看出。

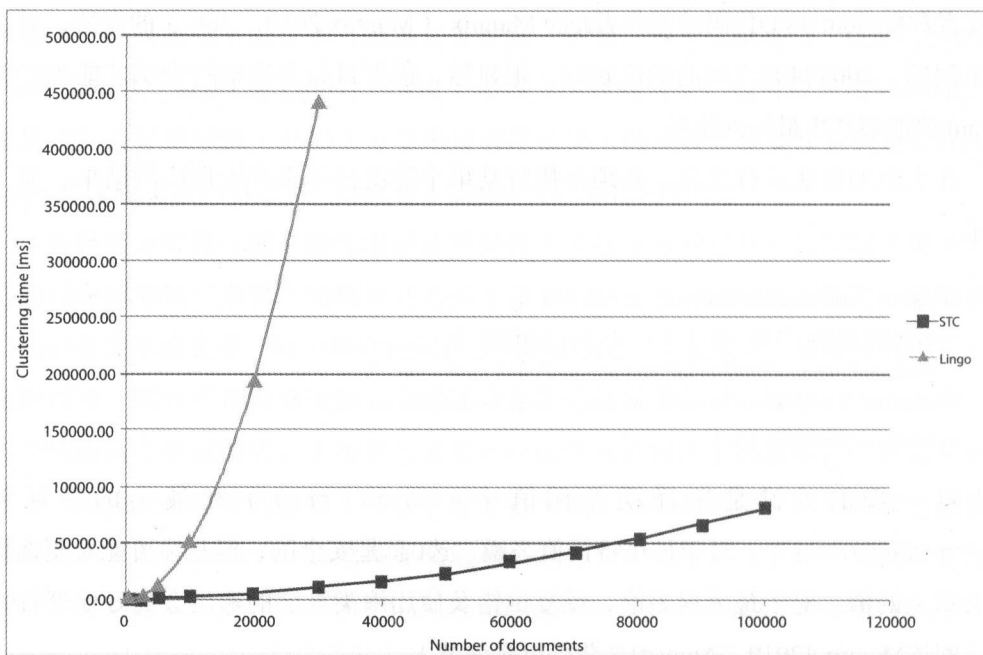


图6-3 STC和Lingo算法在ODP数据 (<http://www.dmoz.org>) 上不同文档数目下的对比

正如在图中可以看到的那样，Lingo比STC要慢很多，但是对结果规模较小的情况，为获得更高的标签质量，可能值得付出更长的运行时间。此外，记住Carrot²可以链接一些原始的矩阵库代码来帮助加快Lingo矩阵分解的速度。对于更重视性能的应用来说，我们推荐使用STC。而如果质量更重要，则选择Lingo。在两种情况下，要花些时间来具体确定哪些属性对数据最有用。要了解Carrot²的完整属性，请参考<http://download.carrot2.org/head/manual/index.html#chapter.components>。

6.7.3 Mahout基准聚类算法

Mahout最强的一个属性是可以将计算分布在计算机网格中运行，这一点要归功于Apache Hadoop的使用。为展示这一点，在Amazon弹性MapReduce (<http://aws.amazon.com/elasticmapreduce/>)上运行K-Means和其他聚类算法，而EC2实例使用不断增长的实例（机器）数来检查Mahout的扩展性。

准备

如6.5.1节所述，邮件存档数据必须要转换成Mahout向量。该准备步骤可以在你本地工作站进行，并不需要Hadoop集群。Mahout发布版中的`prep_asf_mail_archives.sh`脚本（在`utils/bin`目录下）执行如下工作。

- 从`s3://asf-mail-archives/`下载文件并利用tar进行解压；
- 利用基于Mahout `seqdirectory`工具（参考Mahout源码中的`org.apache.mahout.text.SequenceFilesFromMailArchives`）的一个定制工具，将包含gzipped格式的邮件存档数据的解压目录转换为Hadoop `SequenceFile`。每个文件包含多个邮件消息，我们利用正则表达式从每个消息中抽取标题和正文文本。由于邮件的其他消息头能够为聚类提供的信息很少，因此跳过这些信息。每条消息添加到一个块压缩的`SequenceFile`中，最后得到6 094 444个键/值对，存储在283个文件中，这些文件所占的磁盘空间为5.7GB左右。

Hadoop搭建 本节当中我们基于Amazon EC2，使用Hadoop 0.20.2版上的Mahout 0.4版运行所有的基准任务。具体地，我们使用Hadoop发布版中`contrib/ec2`脚本部署的EC2xlarge实例。我们为每个节点指派了3个Reducer（`mapred.reduce.tasks = n*3`），其中每个子进程的最大堆为4GB（`mapred.child.java.opts=-Xmx4096M`）。Hadoop `contrib/ec2`脚本指派了一个额外的主节点（NameNode），该节点在计算集群的规模时并没考虑在内，即一个4节点的集群实际包含5个运行的EC2实例。如何搭建一个Hadoop集群来运行Mahout的详细说明可以参考Mahout Wiki，其地址为：<https://cwiki.apache.org/confluence/display/MAHOUT/Use+an+Existing+Hadoop+AMI>。

内容向量化

`SequenceFile`必须要利用Mahout的`seq2sparse` MapReduce作业转换为稀疏向量。

这里选择稀疏向量是因为大部分邮件消息都较短而所有消息中的独立词项数目又较多。使用默认的seq2sparse配置会产生几百万维的向量，其中 n 维向量中的每一维对应语料库中的一个独立词项。对这么高维度的向量进行聚类是不可行的，又因为邮件存档数据库中长尾独立词项的存在，这样做不太可能产生有用的结果。

为减少独立词项的数目，我们开发了一种定制的Lucene analyzer，它比默认的StandardAnalyzer更积极。具体地，MailArchivesClusteringAnalyzer使用了一个更广的停用词表，去掉了非字母数字的词条并应用了porter词干还原算法。我们也利用了seq2sparse提供的多个特征约简选项。下面的命令给出了如何开始向量化作业的过程：

```
bin/mahout seq2sparse \ --input
s3n://ACCESS_KEY:SECRET_KEY@asf-mail-archives/mahout-0.4/sequence-files
/ \
--output /asf-mail-archives/mahout-0.4/vectors/ \
--weight tfidf \ --minSupport 500 \ --maxDFPercent 70 \
--norm 2 \ --numReducers 12 \ --maxNGramSize 1 \
--analyzerName org.apache.mahout.text.MailArchivesClusteringAnalyzer
```

对于输入，我们使用Hadoop的S3原生协议（s3n）直接从S3读取SequenceFile。注意一定要在URL中包含你的Amazon 访问密钥和秘密访问密钥，这样Hadoop才能访问asf-mail-archives桶。如果你对这些密钥值不太清楚的话，请参考Mahout Wiki上的EC2页。

作者注 由于恶意用户滥用的可能性，asf-mail-archives桶不再存在。这是在Mahout 0.4上用于生成性能指标的桶，出于历史精确性的考虑，这里仍然保留了上述命令。另一个原因是，我们在Amazon上的配额已经用完，而这些基准测试程序的开销比较昂贵！为产生相似的结果，你可以使用Amazon的包含新版本ASF公共邮件库的公开数据集。这些数据集的地址为：<http://aws.amazon.com/datasets/7791434387204566>。

大部分参数已经介绍过，因此我们将集中关注那些对聚类而言十分重要的参数。

- `--minSupport 500`：去掉在文档集中出现次数不到500的词项。对于小语料库来说，500可能太高以致排除重要的词项。

- `--maxDFPercent 70`: 去掉出现在70%或更多文档的词汇, 这有助于去掉那些与邮件相关的词汇, 在文本分析时不考虑这些词汇。
- `--norm 2`: 向量使用2范式进行归一化, 这是因为后面在聚类中使用余弦距离计算相似度。
- `--maxNGramSize 1`: 只考虑单个词汇。

利用这些参数, `seq2sparse`会在40分钟左右在一个4节点集群上创建6 077 604个20 444维的向量。这里产生的向量个数与输入的文档数目不同, 这是因为空向量从`seq2sparse`输出结果中被去掉。运行上述作业之后, 结果向量和词典文件被拷贝到一个公共的S3桶, 因此我们不必在每次运行聚类作业时重建这些向量。

我们也使用了2元组(`--maxNGramSize=2`)进行实验。不幸的是, 这会使向量太大, 达到大概380K维。此外, 建立词汇之间的搭配会显著影响`seq2sparse`作业的性能: 该作业有大约2小时10分钟的时间用于创建2元组, 而又花费至少前面一半的时间来计算搭配。

K-Means聚类基准测试

为开始聚类, 需要利用Hadoop的`distcp`作业将向量从S3拷贝到HDFS, 同前面一样, 我们利用Hadoop的S3原生协议(`s3n`)从S3中进行读取。

```
hadoop distcp -Dmapred.task.timeout=1800000 \
    s3n://ACCESS_KEY:SECRET_KEY@BUCKET/asf-mail-archives/mahout-0
    .4/sparse-1-gram-stem/tfidf-vectors \
    /asf-mail-archives/mahout-0.4/tfidf-vectors
```

该过程只需要几分钟, 具体时间取决于你的簇的大小, 如果在默认的us-east-1区开始你的EC2集群, 那么这里不会产生数据传输的费用。当数据拷贝到HDFS之后, 可以利用下列命令开始Mahout的K-Means作业。

```
bin/mahout kmeans \ -i /asf-mail-archives/mahout-0.4/tfidf-vectors/ \
    -c /asf-mail-archives/mahout-0.4/initial-clusters/ \
    -o /asf-mail-archives/mahout-0.4/kmeans-clusters \
    --numClusters 60 --maxIter 10 \
    --distanceMeasure org.apache.mahout.common.distance.CosineD
    istanceMeasure \
    --convergenceDelta 0.01
```

该作业利用Mahout的RandomSeedGenerator在一开始创建了60个随机中心，这在单独的主节点（这不是一个分布式MapReduce作业）上运行了大概9分钟。该作业完成之后，只要k不变，我们就可以将初始的簇拷贝到S3来避免每次运行基准测试作业时的重建。对于convergenceDelta，我们取的是0.01，这样可以保证K-Means作业为达到基准测试目标至少会有10次迭代，在10次迭代最后，60个簇中的59个收敛。如6.5.2节所述，可以使用Mahout的clusterdump工具来观察每个簇排名最高的那些词项。

为确定Mahout K-Means MapReduce实现的扩展性，我们分别在2、4、8和16节点的集群上运行K-Means聚类作业，每个节点都有3个Reducer。在执行中，平均负载保持健康状态（<4），节点之间不必互换。图6.4给出了我们期望的近似线性的结果。

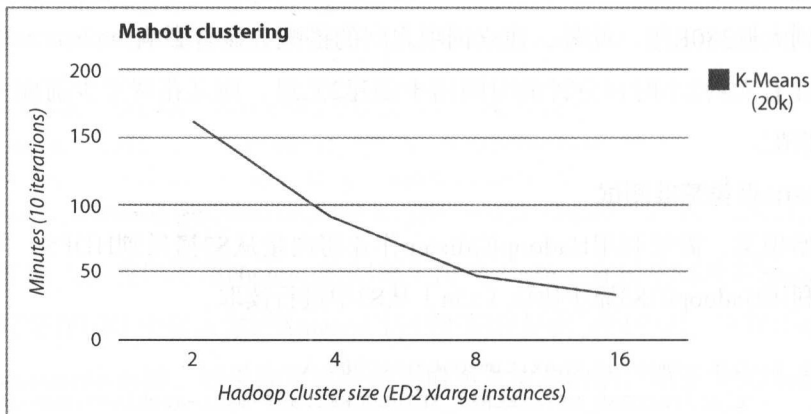


图6-4 Amazon EC2上2节点到16节点集群上运行Mahout的K-Means算法的性能结果图

每次我们将节点的数目翻倍，我们会看到处理的时间大致减半。但是随着节点数的增加，曲线稍微有些变平。之所以造成这种凸形的局面是因为某些节点收到了要求更高的样本，其他节点必须要等待它们结束。因此，某些资源并没充分使用，并且节点数目越多，这种现象发生的可能性更大。此外，当遇到两种情况时，可以想象文档样本之间会出现差异。首先，向量采用稀疏结构来表示。其次，数据集处理长尾特征，这会造成需要计算大型向量的局面出现。上面两种情况在我们的设置中都可以满足。我们也试图在一个4节点EC2large实例集群上运行上述相同作业，其中每个节点带两个Reducer。在这种配置下，我们预期作业会在大约120分钟完成，但是它实际花费137分钟，系统平均负载始终在3以上。

对Mahout的其他聚类算法进行基准测试

我们也对Mahout的其他聚类算法进行了实验，包括模糊K-Means、Canopy和Dirichlet。总体来说，我们不可能在当前数据集上产生任何结论性的结果。例如，模糊K-Means算法中的一次迭代可能比K-Means中的一次迭代要慢10倍。但是人们认为模糊K-Means比K-Means的收敛速度快，其可能需要的迭代次数少。模糊K-Means和两个K-Means算法的变种的运行时间对比如图6-5所示。

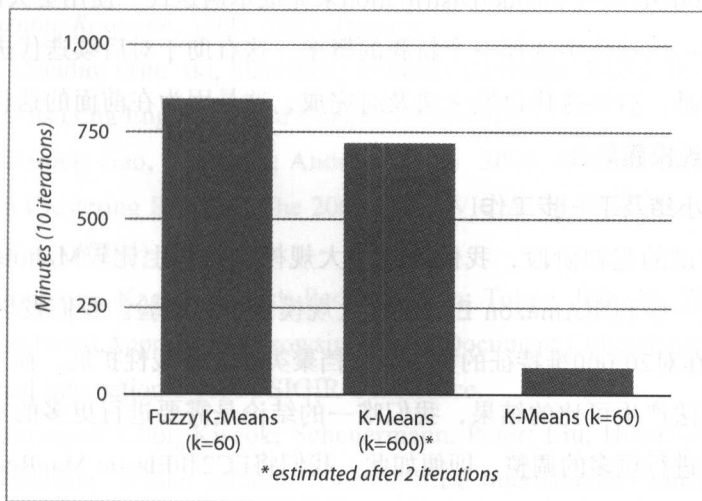


图6-5 不同聚类算法的运行时间对比

我们在实验中使用了四个特大的实例。对于60个簇、平滑参数 m 设为3的情况下，完成所有10遍模糊K-Means的迭代过程需要花费14小时（848分钟）的时间。而相同条件下的K-Means只需要91分钟，大约是前者的十分之一。我们观察到，两个聚类算法的第一次迭代要比后续迭代总是要快很多。此外，第二、三及后续每次迭代所花的时间大致相当。因此，对于不同级别的 k ，我们可以使用第二次迭代的反馈来估计全部10次迭代的时间。当 k 增加10倍，可以预期下降的速度与此成正比。精确地说，我们估计 $k=600$ 时的运行时间为725分钟，这比 10×91 要少点，原因是增加 k 会更好地利用固定开销处理的成本。第一次迭代和第二次迭代之间的区别在于，第一次迭代中随机向量用作簇中心。而在后续迭代中，中心向量更加稠密，需要的计算时间更长。

Mahout的Canopy算法作为一个预处理步骤用于确定大规模数据集的簇数目时可

能很有用。利用Canopy，用户只需要定义两个阈值，这些阈值会影响所创建簇的距离。我们发现，当 $T1=0.15$ 、 $T2=0.9$ 时，Canopy会输出非平凡的结果簇集合。但是寻找这些值所需的时间看起来并不会在加速其他算法时得到补偿。需要记住的是，在本书写作时，Mahout仍然处于pre-1.0发布版，因此，当更多用户使用代码时可以进行加速。

在利用Dirichlet时也会遇到问题，但是利用Mahout社区的辅助，我们能够使用 $\alpha=50$ 、 $\text{modelDist}=L1\text{ModelDistribution}$ 来完成单遍迭代。使用更大的 α 值会有助于增加第一次迭代中选择一个新簇的概率，这有助于对后续迭代进行负载均衡处理。不幸的是，后续迭代仍然无法及时完成，这是因为在前面的迭代中太多数据点被分配给小规模簇集合。

基准测试小结及下一步工作

在基准测试的起初阶段，我们希望在大规模文档集上比较Mahout的不同聚类算法，并产生一些利用Amazon EC2进行大规模聚类的经验。我们发现，Mahout的K-Means实现在对20 000维特征的数百万文档聚类时能够线性扩展。而对于其他聚类算法而言，无法产生可比的结果，我们唯一的结论是需要进行更多的实验，以及要对Mahout代码进行更多的调整。即便如此，我们将EC2和Elastic MapReduce的搭建说明写到Mahout Wiki上以便其他人能够基于我们的工作开展自己的工作。

6.8 致谢

在撰写本章时Ted Dunning、Jake Mannix、Stanisław Osin'ski和Dawid Weiss提供了有价值的意见，在此对他们表示感谢。Amazon Elastic MapReduce和EC2上Mahout基准测试可能归功于来自Amazon Web Services Apache Projects Testing Program的配额。

6.9 小结

不管是要减少通读的新闻量，还是快速对歧义搜索词项进行概括总结，或者是在大型文档集中识别主题，聚类都是一种能在你的应用中提供有价值的发现功能的高效方法。本章中，我们讨论了聚类背后的很多概念，包括一些用于选择和评估聚类方法的因素。然后我们展示如何使用Carrot²和Apache Mahout来完成搜索结果、文

档和词到主题的聚类,从而集中关注真实世界中的例子。本章最后考察了一些提高聚类性能的技术,其中包括使用Mahout的奇异值分解代码。

6.10 参考文献

Blei, David; Lafferty, John. 2009. "Visualizing Topics with Multi-Word Expressions." <http://arxiv.org/abs/0907.1013v1>.

Blei, David; Ng, Andrew; Jordan, Michael. 2003. "Latent Dirichlet allocation." *Journal of Machine Learning Research*, 3:993–1022, January.

Carpineto, Claudio; Osin'ski, Stanisław; Romano, Giovanni; Weiss, Dawid. 2009. "A Survey of Web Clustering Engines." *ACM Computing Surveys*.

Crabtree, Daniel; Gao, Xiaoying; Andreae, Peter. 2005. "Standardized Evaluation Method for Web Clustering Results." The 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI'05).

Cutting, Douglass; Karger, David; Pedersen, Jan; Tukey, John W. 1992. "Scatter/Gather: A Cluster-based Approach to Browsing Large Document Collections." Proceedings of the 15th Annual International ACM/SIGIR Conference.

Dash, Manoranjan; Choi, Kiseok; Scheuermann, Peter; Liu, Huan. 2002. "Feature Selection for Clustering - a filter solution." Second IEEE International Conference on Data Mining (ICDM'02).

Dash, Manoranjan, and Liu, Huan. 2000. "Feature Selection for Clustering." Proceedings of Fourth Pacific-Asia Conference on Knowledge Discovery and Data Mining.

Dean, Jeffrey; Ghemawat, Sanjay. 2004. "MapReduce: Simplified Data Processing on Large Clusters." OSDI'04: 6th Symposium on Operating Systems Design and Implementation. http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf.

Deerwester, Scott; Dumais, Susan; Landauer, Thomas; Furnas, George; Harshman, Richard. 1990. "Indexing by latent semantic analysis." *Journal of the American Society of Information Science*, 41(6):391–407.

Dunning, Ted. 1993. "Accurate methods for the statistics of surprise and coincidence." *Computational Linguistics*, 19(1).

Google News. 2011. <http://www.google.com/support/news/bin/answer.py?answer=40235&topic=8851>.

Liu, Tao; Liu, Shengping; Chen, Zheng; Ma, Wei-Ying. 2003. "An evaluation on

feature selection for text clustering.” Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003).

Manning, Christopher; Raghavan, Prabhakar; Schütze, Hinrich. 2008. *An Introduction to Information Retrieval*. Cambridge University Press.

Mannix, Jake. 2010, July. “SVD Memory Reqs.” http://mail-archives.apache.org/mod_mbox/mahout-user/201007.mbox/%3CAANLkTik-uHrN2d838dHfYwOhxHDQ3bhHkvCQvEIQCLT@mail.gmail.com%3E.

Mannix, Jake. 2010, August. “Understanding SVD CLI inputs.” http://mail-archives.apache.org/mod_mbox/mahout-user/201008.mbox/%3CAANLkTi=ErpLuaWK7Z-2an786v5AsX3u5=adU2WJM5Ex7@mail.gmail.com%3E.

Steyvers, Mark, and Griffiths, Tom. 2007. “Probabilistic Topic Models.” *Handbook of Latent Semantic Analysis*. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.9625&rep=rep1&type=pdf>.

Zamir, Oren, and Etzioni, Oren. 1998. “Web document clustering: a feasibility demonstration.” Association of Computing Machinery Special Interest Group in Information Retrieval (SIGIR).

第7章 分类及标注

本章内容

- 学习分类及标注的基本概念
- 了解分类在文本应用如何使用
- 利用开源工具构建、训练和评估分类器
- 将分类集成到搜索应用中
- 在已标注数据上训练得到标签推荐引擎

你可能已经在所访问的网站某处遇到过关键词标签。照片、视频、音乐、新闻、博文或推文常常也带有对当前浏览内容进行快速描述的词和短语，同时也包含对相关对象的链接。你可能见过关键词云，即采用不同大小的词来展示某人喜欢的讨论主题、影片类型或音乐风格。标签在Web中无处不在，它们用于辅助浏览或者组织从新闻到书签的所有对象（参考图7-1）。



图7-1 一篇推文中所用的标签。以#号开始的Hashtag用于标识推文中的关键词，以@号开始的标签指向其他用户

标签是有关数据的数据，或者也称为元数据。它们可以应用任意类型的内容，并以非结构化形式存在，从简单的关键词或用户名列表到高度结构化的属性，如身高、体重和眼睛颜色等。

如何构建这些标签？有些标签是作为编辑流程的一个环节而生成的。作者或管理者会赋予一些描述性的词项，这些词项可能是第一时间想到的词，或者是从一个被认可的词和短语集合中精心选出的词。其他情况下，标签由网站用户给出。每个人按照自己的观点选择合理的词项给对象打标签。不论是一本书还是一首歌，成千上万的个体会根据他们的视角来对其定义一段内容，集体的智慧或愚蠢会最终获胜。

机器学习允许你从内容中自动或半自动生成标签。算法用于观察已有标签的标注过程，并对已有标签推荐其他标签，或者为新的未标注数据推荐标签。这种自动打标签的形式是分类的一个特例。

分类面临的挑战可以简单概括为：给定对象，如何将它分配到一个或者多个预先定义的类别中？为解决这个挑战问题，必须要考虑对象的性质。它和其他对象相似程度怎么样？如何将具有共同性质的对象分到一组而排除性质不同的对象？

分类算法通过例子来学习，其使用那些通过人工或某个自动过程组织为类别的数据。通过训练过程，分类算法确定那些能够标志对象属于某个给定类别的性质或特征。训练之后，分类算法能够对未标注数据分类。

第6章介绍了另一种称为聚类的学习算法。分类和聚类可以看成同一硬币的两面。两类算法都试图通过使用那些能够确定类别归属的特征来将标签分配给对象。分类和聚类的不同之处在于，前者使用一个预先定义的标签集合，能够学习对象和这些标签吻合的程度。这种方法称为有监督学习，其中分类算法分配的标签基于外部输入（如用户定义的类别集合）而形成。聚类是无监督学习的一种，并不使用预先定义的标签集合。聚类基于共同的特性对对象分组。尽管分类和聚类存在上述区别，分类和聚类算法都可以用于进行文档归类。

文档分类是指将类别或主题相关的标签分配给文档的行为。这是分类算法的一个应用。在文档分类中，开始有一些训练文档样例，每个样例都分配到一个或多个类别或主题领域中。分类算法构建一个模型，该模型刻画了词项和其他文档特征（如长度或结构）如何与主题相关。模型构建之后，可以用它来分配主题领域，即

将文档分类。

本章一开始将会对分类进行概述，并讨论生产系统中分类器的训练过程。从那里开始，我们会考察一些不同的分类算法，并学习它们如何自动对文档categorize和对文档打标签。其中有一些算法基于统计模型，比如朴素贝叶斯和最大熵分类器，而有些技术使用了第3章所介绍的信息检索中使用的向量空间模型，比如kNN和TF-IDF分类器。

这些算法会通过多个实际动手的例子来介绍，这些例子会使用命令行工具和开源项目的代码来完成，这些开源项目包括OpenNLP、Apache Lucene、Solr和Mahout等。每种情况下，都会一步步介绍分类器的创建过程。我们会探讨获取和准备训练数据的方法，利用每种算法来训练分类器并了解如何对输出结果的质量进行评估。最后，我们会展示分类器集成到生产系统的方法。每个展示过程中，我们会给出命令和需要关注的代码。到本章结束之时，你将能够根据自己的目标来对这些介绍过的例子进行定制，并对你的应用构建自动的分类器和标签推荐系统。

7.1 分类及归类概述

在计算意义上说，分类是将标签赋给数据的过程。定义对象的特征集合，分类器试图将标签分配给该对象。分类器通过从其他已经标注的对象中获得知识来实现上述过程。这些例子称为训练数据，是分类器用于对未标注对象进行决策的先验知识源。

归类是分类的一个特例。它将类别分配给对象。其他分类算法可能基于输入简单地是/否二类决策，比如某个欺诈检测器能够表明信用卡交易存在欺诈。分类算法可以将对象放入一个较小的类别集合，比如将汽车归为小轿车、厢式轿车、越野车和大蓬货车。本章当中讨论的很多概念总体上都属于分类，其他一些概念与归类更相近。你会看到有些术语可以互换¹。

本章所讨论的文档归类是指将类别赋予文本文档的过程。在这里的例子中，我们将一些基于主题类别赋给文档，但是其他一些文档归类应用可能并非如此，比如情感分析利用文档归类来确定产品评论的正负倾向，或者邮件或客户请求背后的

¹ 在文本分类中，一般并不区分classification和categorization。——译者注

情绪。

为理解自动分类的过程，设想区别直升机和一般飞机的过程。可能没人会明确地告诉你直升机有别于一般飞机的特征是“水平旋转的螺旋桨”或“没有固定翼”。在考察每种飞机的多个样例之后，你能够区别哪是一般飞机哪是直升机。你能够无意识地从所谓的直升机中抽取特征，然后利用这些特征来识别其他的直升机并确定那些带有发动引擎的东西不是直升机。当看到一个带有水平旋转桨的飞行器，你可以马上认出这是一架直升机。分类器算法的工作流程与此类似。

上例也触及确定对象类别之间差异的特征选择的重要性。不论是直升飞机还是一般飞机都能飞行并且运送旅客。这两个特征对于区分直升机和一般飞机毫无作用，因此利用这些特征来训练分类器将毫无意义。在上一段的例子中，假设直升机是黄色的而飞机是蓝色的。如果对于每一种类别你只能看到这种样例，你可能会想所有飞机都是蓝色的而所有直升机都是黄色的。但是你的实际经验让你很清楚，对于判断这两种飞机类别颜色根本无关紧要。如果没有这种知识，那么在分类器决策时考虑颜色的分类算法会出错。这凸显了算法在广泛多样的数据上进行训练的重要性，这些数据要尽可能覆盖更多可能的特征及其组合。

分类算法通过例子来学习，使用那些通过人工或者自动组织成类别的训练数据。通过观察特征和类别之间的关系，算法能够学习到哪些特征对于确定正确的标签很重要，而哪些特征对对象赋予标签的过程提供很少甚至误导性的信息。训练过程的结果是一个可以用于对未标注数据进行分类的模型。分类器考察待分类对象的特征，并使用模型来确定每个对象的最佳类别。根据分类算法的不同，分类器可能输出单个或者多个标签，每个标签都带有一个得分或者概率，该值给出的是该标签相对于其他标签的一个序值。

目前存在很多类型的分类算法。一种区别方式是看算法的输出。输出两个离散值（如是/否）的算法称为二类分类算法。而其他算法支持多种输出，结果可能是一个离散类别或连续值集合（比如浮点数得分或概率）。

二类分类器输出的结果能够标识待计算对象是否属于某个类别。这类分类器的一个最简单的例子是垃圾邮件过滤器。垃圾邮件过滤器会分析邮件中的特征并确定该邮件是否垃圾邮件。我们在7.4节讨论的贝叶斯分类算法开发了一个统计模型来确定对象是否属于某个类别，该算法正好是实际当中常用于垃圾邮件检测的算法。支

持向量机 (Support Vector Machine, 简称SVM) 也是一个二类分类算法, 它试图寻找一条直线或称为超平面的 n 维平面来将一类样本的特征空间与另一类样本的特征空间分开。

有时, 可以组合二类分类器来进行多类分类。多个二类分类器中的每一个都分配一个类, 输入会在每个类上进行评估以确定它属于哪个类。根据算法的不同, 输出可以是输入最可能隶属的单个类别或者是输入隶属的多个类别, 且每个类别都有一个权重来描述输入对象隶属于该类别的相对可能性。本章后面介绍的Mahout贝叶斯分类器就是一个训练多个二类分类器的例子, 这样就可以在多类分类机制下分配类别。

多个二类分类器有时会组织成类似树的结构。在这种情况下, 一篇文档隶属于类A, 而A又有B和C两个子类别, 该文档会分别在B、C这两个类训练出的分类器中进行评估。如果该文档属于B类, 则它会与B的子类别E和F进行匹配。如果文档最后不能分配给E和F类, 那么会将它归于B类, 否则的话文档会归属于与之匹配的最低层的叶子类别。当类别本身就是层次型结构时 (如分类目录), 上述方法很有用。这种层次方法的变形已经在诸如层次型二类决策树和随机森林的方法中取得了很好的效果。

7.5节将要介绍的最大熵文档分类器就是多类分类算法的一个例子。该分类器将文档中的词作为特征而将主题领域作为类别。训练过程构建了一个将词和主题关联起来的模型。对于一个未分类文档, 上述模型用于确定特征的权重然后将它们最终用于生成文档可能属于的主题领域。

7.8节中会讨论一些分类算法, 它们利用了第3章所介绍的向量空间模型的性质。在这些方法中, 所有的已分类文档之间的距离会与未分类文档的距离进行比较, 该结果用于确定此文档所属的类别。在这种上下文中, 未分类文档就变成查询, 该查询用于返回已分类的文档或者表示每个类别内容的文档。本章当中我们会讨论这个方法, 也会在一个利用Lucene来索引训练文档并对给定查询返回文档的例子中来介绍这个方法。

有大量的分类算法和工作适用于文档分类或标注任务。本章主要介绍一些容易通过开源项目代码来实现的算法。这些例子可以作为进一步探索分类技术的起点。不管这里介绍的方法或算法如何, 这里所讨论的大部分内容都与分类和其他监督学

习任务有关。我们将通过一系列贯穿本章的例子探讨多个关注点，比如收集训练数据、识别特征集合以及评估分类质量等。每个例子会给出一个面对分类或标注的不同方法，但是这些例子建立在先前讨论的例子中，这些例子包括本章的例子和本书其他节的例子。

7.2 分类过程

不管所用的算法如何，开发自动分类器遵循一个基本通用的过程。如图7-2所示，该过程由多步组成：准备、训练、测试和生产。为了对分类器调优而产生最优结果，通常上述过程会反复迭代，每一步可能也会自动或人工重复多次。每一步的反馈都有助于确定在准备和训练阶段如何修改来获得更好的结果。这里说的调优的意思是，利用测试阶段的结果来提高训练的效果。在分类器用于生产环境之后，往往必须要将该分类器扩展到训练数据不包含的其他情况上去。

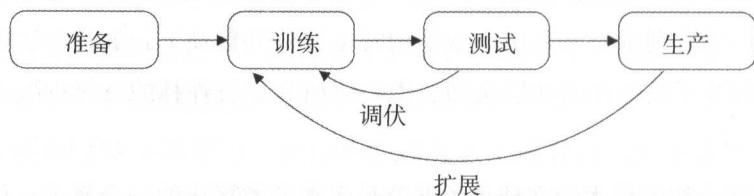


图7-2 开发自动分类器的几个阶段：数据准备、模型训练、模型测试及在生产中部署分类器。根据测试结果调整训练过程，对分类器进行扩展以覆盖部署之后出现的新样例

准备阶段涉及训练阶段数据的准备过程。这里需要选择分类器的目标标签集合、训练所用的特征的识别方式以及数据集中用于测试的留存数据。当这些决定都已经确定时，这些数据必须要转换成训练算法所使用的格式。

数据准备好之后，就进入训练阶段，训练算法会处理每个已标注数据并确定特征与该标签的关联方法。每个特征都与分配给该文档的标签关联，训练算法会对特征和类别标签的关系进行建模。有大量算法可以用于训练分类模型，但是最终，训练算法会识别那些对于区分不同类别数据重要的特征以及不太能够区分不同类别的特征。通常而言，分类算法会接受那些能够控制模型建模方法的参数。很多情况下，会对这些参数的最优值给出有一个有根据的猜测结果，然后通过一个迭代过程反复调整。

在测试阶段，分类算法会在一些额外的称为测试数据的数据上进行评估。该评估过程将每个样本真正属于的类别与分类器分配的类别进行比较。这些正确和错误的分配结果用于确定训练算法的精度。有些算法也会从训练阶段产生一些输出结果来帮助理解算法对训练数据的诠释过程。然后，该输出结果会作为反馈信息来调整准备阶段和训练阶段的参数和技术。

在整个生命周期内，分类器可能会被训练多次。在对分类器进行初始开发时，一种很常见的做法是重复训练和测试阶段，每次重复中调整训练过程的目的都是为了得到更好的结果。这些调整可能涉及通过增加和删除样例对训练数据进行轻微调整、对从数据中抽取特征的方法进行修改、修改目标类别或者修改对学习算法行为进行控制的参数。很多分类算法（比如7.5节要讨论的最大熵算法）在训练过程中自动迭代直到收敛到最佳答案为止。而很多其他分类方法的迭代过程是手工完成。某些情况下，迭代的训练过程被并行化，一个分类器的多个变体在同一时间进行训练，能够产生最优结果的分器被选出来放入生产环境中。

当分类器放入生产环境下以后，通常它后来需要重新训练以扩展所处理领域的知识。当新词汇随着时间的推移不断出现并且在区别不同类别中扮演主要角色时，上述情况往往就会发生。例如，我们来看看一个将产品评论按照主题领域进行组织的分类器的例子。当新产品发布时，必须要看看包含产品名称的已标注文档，否则就无法明白 *android* 很可能表示智能手机而 *ipad* 表示移动计算设备。

迄今为止，我们已经对分类过程各个步骤中的应用知识有所了解，接下来我们将深入讨论每一步中需要考虑的问题。

7.2.1 选择分类机制

分类算法通过实例使用数据学习，这些数据事先已经通过手工或自动方法组织成类别。分配给对象的类别都有一个有别于分类对象的名字和意义。每个类别都在其他类别的上下文中存在，该上下文称为分类或归类机制的一个体系。有些分类机制可能很严格，在该机制下每个对象只能属于一类。其他分类机制则相对灵活一些，它肯定真实世界的对象都存在不同方面这个事实。有些分类机制比如林奈式动植物分类机制（Linnaean taxonomy）可能具有严格的层次意义。而其他一些分类机制可能没有除了语言学关系之外的特别结构，比如Flickr或Technorate上的简单标签关键

词。分类机制之间的范围可能相差很大。它们可能覆盖一个很广的主题领域，比如图书馆中使用的杜威十进分类法就是如此，也有可能覆盖的领域很窄，只针对特定的领域，比如用于描述残疾人技术辅助器具的分类机制。

在很多场景下，分类机制是一个有机系统并不断演化。在诸如社会化标签网站 delicious.com 的情况下，标签通过其用于网页分类的方式来定义。用户选择那些描述网页的词，这种行为一天会发生几百万次。分类机制中所用的词汇经常会演变，其意义也会基于其使用方式而有可能持续改变。随着时间的推移，分类机制自底向上出现并变化，这与自顶向下的分类方法正好相反，在后者当中，一个预先可能主题层次集合可供用户选择。

决定应用中分类机制的选择实际上是一种取舍过程。自底向上的基于标签的机制以简单换取正确率，但是当标签一词多义或者多词一义或者甚至遇到一个更简单的情况——用户使用单词复数来标注资源而另一个用户用单数形式来搜索时，上述机制可能会纠缠于语言用法问题。而自顶向下的机制可能不会有这样的问题，可以提供类别空间下的权威表示，但是在遇到新词汇或意义时可能难以调整。

7.2.2 识别文本分类中的特征

第2章当中，我们讨论了从文本抽词的不同方法，这些词可以设想为文本的特征，用于后续处理过程。这些词在分类算法中作为特征使用，用于确定包含这些词的文档出现在哪个类别中。

最简单的一种方法称为词袋方法，它将文档看成词的集合。出现在文档中的每个词会被看成特征，这些特征会按照其出现的频率计算权重。为了基于词在训练语料库中的权重来为不同词赋予不同的重要程度，可以利用第3章介绍的TF-IDF机制对每个词生成权重。根据训练语料库规模的不同，利用该语料库中的词项子集来构建分类器可能是必须的。去掉那些过于频繁或者IDF很低的词会让你在语料库中最重要的那些词上训练分类器，这些词是能够区分不同类别的最有力的词。第3章还介绍了一系列其他的权重机制，这些权重机制可以用于选择部分特征进行分类决策。

词组合起来往往能够得到有用的文档特征。与将文档中出现的词作为特征不同，可以利用 n 元组来获得最重要的词语组合。一个类别往往包括此类别独有的词语

组合, 比如*title insurance*、*junk bond*或*hard disk*等。选择语料中所有的词语组合会导致特征爆炸, 但是算法能够识别出统计上相关的称为搭配的词语组合, 并去掉那些没有什么价值的词语组合。

除去内容之外, 在构建分类器时其他文档特征可能有用。构成语料库的文档可能具有提高分类器算法质量的性质。比如作者和信息源这种文档的元数据往往十分有用。文档刊登在日文报纸上这个事实会暗示文档更可能属于Asisan Business类。某些作者可能常常撰写体育类文章, 而其他可能撰写技术类文章。文档的长度可能在确定到底是学术论文、邮件消息或推文时会成为一个判断因素。

也可以借用一些额外资源来从文档中导出特征。通过像WordNet一样的词汇资源来以通过加入文档关键词项的同义或近义词实现词项扩展, 扩展后的词项会作为特征使用。实体往往会通过第5章所描述的算法来抽取并加入到特征中, 因此Camden Yards或Baltimore Orioles可以作为确定文章是否体育类的独立特征来使用。进一步而言, 聚类算法或其他分类算法的输出可以输入到一个分类器中用于确定类别归属。

既然选择项这么多, 那么到底哪儿才是最佳的起点? 你可以利用标准向量模型将词袋方法和TF-IDF权重机制综合在一起使用。在本章的例子中, 我们会从此开始, 并在介绍过程中给出多种特征选择方法。算法在自动分类系统中扮演着重要的角色, 但是不管选择的算法如何, 特征选择可能决定最后结果的成败。

7.2.3 训练数据的重要性

分类器的精度取决于训练所用特征及训练文档的质量和数量。如果训练样本的数量不够的话, 分类器无法确定特征和类别之间是如何关联的。在这种数据量不够的情况下训练过程会就上述关联关系作出错误的假设。这样就不能区分两个类别, 或者不完整数据可能意味着特征与某个特定类别相关但实际并不相关。例如, 很直观就会知道在确定某个对象是直升机还是一般飞机时颜色并不是区别特性, 但是如果分类算法只看到黄色的飞机样本而没有看到黄色的直升机的话, 可能就会认为所有的黄色飞行器都是飞机。一个全面均衡的尽可能包含更多相关特征的训练集以及一定数量的训练样本对于产生一个精确的模型相当重要。

但是这些训练数据到底从哪里来? 一种方法是手工将类别分配给数据。像路透社(Reuters)一样的新闻机构投入了大量的时间精力来手工标注经过它们的新闻报

道。数百万delicious.com用户的手工标注结果可以作为已标注网页源来使用。

也可以利用自动过程来获得训练数据。在7.4节的Mahout贝叶斯例子中，我们会讨论如何通过关键词搜索来获得一系列与某个主题领域相关的文档。一个或多个关键词与某个类别关联，然后使用搜索来返回包含这些关键词的文档。这种称为自举的过程能够获得相当精度的分类器。

互联网上存在大量可用数据可以用于训练分类器。诸如维基百科及其相关的一些项目（如Freebase），这些项目可以导出大量可用数据²，这些数据可以提供大量文档库，其中很多文档都有类别和标签或者其他对训练分类器有用的信息。

伴随机器学习的研究，也出现了不少可供使用的测试数据集。当重现已有研究的结果或者比较不同方法的性能时，这些数据集十分有用。这些数据集中的很多仅限于非商业用途并且需要引用，但是它们确实提供了一条很好的方式来使分类器跑起来、探索分类的不同方面并提供了让你知道在正确方向上前行的基准。

最著名的一个测试文档集是RCV1-v2/LYRL2004文档分类测试语料（参考Lewis[2004]），该语料包含超过80万篇的路透社新闻报道。与语料配套的论文深入介绍了该语料并给出了训练方法论以及多个著名文本分类方法的结果。在RCV2发表之前，还有一份叫做Reuters21578的路透社语料可用，该语料也被广泛使用。尽管该语料包含的文件数目明显很少，但是由于基于该语料发表的论文数据很多的原因，该语料仍然可以作为基准文本报道文档集使用。另一个称为20 Newsgroup的测试语料（获取地址为：<http://people.csail.mit.edu/jrennie/20Newsgroups/>）包含了大概11000篇来自20个独立互联网新闻组的文章，其主题覆盖了计算、体育、汽车、政治和宗教等领域，可以作为一个小规模的组织良好的训练和测试语料使用。

Stack Overflow（<http://www.stackoverflow.com>）的母公司Stack Exchange和其他一些社会问答网站，可以在Creatvie Commons许可证（参考<http://blog.stackoverflow.com/category/cc-wiki-dump/>）下进行数据的导出。Stack Overflow中的每个问题都被用户社区的关键词所标注。包含这些标签的数据导出后是一个很好的训练数据源。7.6.1节中将使用该数据的一个子集来构建我们自己的标签推荐系统。

2 维基百科的数据可以从地址http://en.wikipedia.org/wiki/Wikipedia:Database_download导出，而Freebase的数据则可以从http://wiki.freebase.com/wiki/Data_dumps中导出。

如果所需数据无法以批量导出形式获取,但是又在互联网上存在,那么开发一个目标导向的Web爬虫来获得分类器训练所需数据并不少见。有些大型网站比如Amazon甚至提供了一个Web服务API来从该网站获取内容。开源的采集框架(如Nutch和Bixo)均提供了从互联网搜集训练数据的一个很好的起点。这样做的时候,要注意每个网站的版权和服务条款以确保收集的数据可以为自己所用。一定要对网站礼貌访问,限制采集器每隔数秒爬取少量网页,看上去就像为下载数据支付带宽费用一样。访问时只取你所需,除此之外不要下载更多信息,不要把负担强加于给定网站,如将费用强加于网站所有者或对正常用户拒绝服务。做个好公民,有疑问时可以联系网站所有者。这样可能有机会获得非公开的数据。

如果上述方式都不行你需要自己手工标注数据,也别灰心绝望。除了招募你的朋友、家庭、桥牌社成员以及街道上的随机过往人员来为你手工标注Twitter推文之外,你也可以转向土耳其机器人的帮助。土耳其机器人是开发公司将需要集体智能的任务(采用Amazon MT的说法称为human intelligence task, HIT)廉价转给全球用户完成的一种方式。亚马逊土耳其机器人网站包含丰富的构建和执行此类任务的信息。

将人工判定结果作为训练数据

人们在评估或训练计算机算法时有很多提供反馈的方式。在信息检索中,一种普遍的做法是利用人工判定结果来评估文档的相关性。用户对确定返回结果的相关与否并根据这些判定结果对检索结果的质量进行评分。有关人工判定结果问题的深入描述可以参考很多研究工作,其中有些论文在本章结尾处引用。

记住,我们人类并非十全十美。当着手收集大规模人工判定结果的时候,要检查进行判定工作的人是否理解类别或标签的意义,他们是否能够做出一致的判定,并且不会随时间显著改变。通过将某个人的判定结果与他人或机器进行标注的文档进行对比,可以检查人工判定的可靠性以及分类机制的明晰性。通过用户随时间反复判定这个过程可以确定判定的一致性。

大部分分类算法支持一些额外的参数,这些参数作为训练或分类过程执行的一部分影响所执行的运算。诸如朴素贝叶斯的算法参数很少,而其他如支持向量机(SVM)之类的算法拥有很多可修改的参数。通常来说,训练分类器是一个设计多个迭代的过程:基于每个参数的初始值来训练,之后进行评估,然后为获得最佳分

类性能来对这些参数值进行轻度调整。

7.2.4 评估分类器性能

训练好的分类器会对已标注文档进行分类，然后将分类结果和已有结果进行对比从而对该分类器进行评估。分类器的质量通过其产生与正确标注相等的能力来衡量。分类器分配的标签和已有标签匹配的比例称为分类器的精度。该指标虽然提供了分类器的整体意义上的性能，但是需要深入了解遇到的错误的本质。

第3章所介绍的正确率和召回率的变形可以用于产生分类器更细节的度量指标。这些指标基于遇到的错误的类型而产生。

当考虑二类分类器的输出时，存在如表7-1所示的四种基本结果。其中的两个结果表示正确的响应，而另两个结果表示不正确的响应。两个正确的响应中，其中一个是已有标签和分类器给出的标签都表示正例，而另一个则两个标签都表示负例。它们分别成为真阳和真阴。而两个错误的结果中已知标签和给出的标签不一致。

第一种情况称为假阳，其中分类器认为对象属于某类而实际上并不属于该类。另一种情况成为假阴，分类器认为对象不属于某类而实际上却属于该类。在统计中，这两类错误分别称为第一类错误和第二类错误。

在分类中，正确率为真阳的数目除以真阳和假阳的数目之和。而召回率为真阳的数目除以真阳和假阴之和。第三种称为特异度或真阴率的指标为真阴的数目除以真阴和假阳之和。

表 7-1 分类的结果

	属于该类	不属于该类
分配给该类	真阳	假阳（第一类错误）
未分配给该类	假阴（第二类错误）	真阴

根据应用的不同，可能对其中一类错误的敏感度高于另一类错误。在垃圾邮件过滤中，由于假阳会导致用户错过一封正常邮件，所以会带来很高的代价。而尽管在收件箱中看到一封垃圾邮件会令人懊恼，但是该错误很容易通过删除键来弥补，因此此时可能更会接受假阴。根据应用的不同，可能会集中关注整体的精度、正确率、召回率或者特异度。

在多类分类器的情况下，一种很普遍的做法是对分类器可能分到的每个类别都使用上面的每个指标，然后将这些指标聚合，得到整个分类器的某个平均精度、正确率和/或者召回率指标。

除了了解真阳或假阳的数目之外，理解类别之间的相互作用（即分类中互相错分）往往更加有用。一个称为混淆矩阵（confusion matrix）的结果表示方法可以给出每个带标签文档被分配的情况，从而还原错误的本来面目。在分类器错分的情况下，混淆矩阵会给出文档被错分到的类别。某些情况下，大多数错误可能都会涉及将某类文档分配到另一个类别。这些错误可以用于确定训练数据或特征选择策略中存在的问题。

为了计算上述指标，必须要将一些已标注数据从训练过程中留存出来。如果有200篇新闻报道与足球类有关，可以从中选择180篇作为训练集而将另外20篇作为留存数据，这样你就可以保证分类器能够精确地识别足球类。非常重要的一点是永远不要在测试数据上训练，否则你的测试将极度偏斜，产生带有欺骗性的高精度结果。如果遇到结果好得不太真实的话，检查一下以确认没有在测试数据上进行训练。

有多种不同方法将训练数据划分成训练集和测试集。随机选择文档是一个不错的起点。如果数据具有诸如出版日期一样的时间维度，那么按照日期对文档排序并将最新的文档作为测试数据可能十分有用。这样可以检验新来的文档能否基于旧文档中的特征进行精确分类。

在进行分类器评估时，往往不止对训练集和测试集进行单个划分。前面提到的200篇新闻报道的例子可以分成10组每组20篇文档。然后使用不同的组之间的组合方式来训练多个分类器。例如，一个分类器使用第1到第9组数据作为训练集，将第10组作为测试集，而另一个分类器使用第2到第10组数据作为训练集，将第1组作为测试集，其余依此类推。每次测试的结果精度进行平均计算得到最终的度量分类器性能的精度。这种方法称为 k 折交叉校验，其中 k 表示数据划分成的组数目。该方法普遍用于源于统计的分类方法。

除了上述指标和方法之外，其他一些评估方法也可以用于判断分类器的性能。一种称为曲线下面积（area under curve, AUC）的指标在类别之间的训练文档数目不均衡的情况下十分有用。一个称为对数似然率（log-likelihood ratio，有时就称为对数

似然)的指标在评估统计模型比多次训练的结果时有用。

7.2.5 将分类器部署到生产环境

当训练出一个能产生足够质量结果的分类型器之后，必须要考虑如下问题：

1. 将分类器作为更大应用的一部分部署到生产环境。
2. 在生产中更新分类器。
3. 随时间推移不断评估分类器的精度。

本章考察的每个分类器都可以作为更大服务的一个模块进行部署。一个常见的部署模型是将分类器部署为一个长时间运行进程的一部分。启动时模型装入内存，服务会收到分类的请求，每次请求可以是单篇文档或者一批文档的请求。OpenNLP的MaxEnt分类器就采用这种方式运行，启动时将模型加载到内存，然后在其生命周期内反复使用。大型的模型无法全部加载到工作内存，所以必须有部分存在磁盘上。Apache Lucene使用混合的磁盘/内存机制来存储文档的索引，因此它在大模型的情况下也能支持得很好。Mahout贝叶斯分类器支持多种数据存储机制，同时使用内存和基于分布式数据库HBase的存储方式。它也提供了一个API以便能够方便实现适合自己模型的数据存储方式。

分类器部署之后，必须要能在新数据可用时实现对模型的更新。有些模型可以在线更新，而有些模型需要离线构建一个替代模型并将之与现有模型交换。一个能够在线更新的分类器可扩展到能实现对新词汇的处理。Lucene的索引结构使得在不离线处理查询的情况下很方便地实现文档的添加和替换。在其他情况下，比如模型存储在内存中，使用分类器的应用必须能以这种方式开发，及第二个模型必须要在原始模型仍然活跃的情况下加载到内存。当第二个模型的加载完成之后，它会替换原始模型并将原始模型从内存中剔除。

随时间评估分类器的性能涉及收集额外的数据来评估分类器的性能。这可能涉及保留分类器在生产中看到的部分输入并人工选择合适的类别。留心一下分类器发生错误的情况十分有用，但是收集大范围的样本数据来评估更加重要，而不只是简单地考察分类器分错的情况。留心一下新的主题领域或词项或者讨论的话题。不论数据如何，在生产中评估分类器与将分类器作为开发过程的一部分来评估没有什么区别。对你而言，必须仍要有已标注的测试文档，这样你就可以对它们进行分类然

后将分类结果与已有标签进行比较。

通常来说,必须要对分类机制进行修改来容纳新数据。当设计应用时,要考虑上述情况带来的影响。如果存储了分类文档,所分配的类别可能随着分类机制的改变而变得过时。根据应用的不同,可能需要对文档重新分类以适应新机制,为实现这一点必须要访问原始的内容。另一种做法是在旧类和新类之间建立一个映射机制,但是当类别合并和拆分时,这种做法会变得难以维护。

现在已经考察了围绕训练和部署过程的一些问题,下面来实际训练和测试一些分类算法。接下来多节中,我们会考察三种分类方法的变形,并开发出一个标签推荐引擎。贯穿每个例子始终的是,我们会介绍它们当中的准备、训练、测试和开发过程。

7.3 利用Apache Lucene构建文档分类器

有些分类算法称为空间技术。它们利用第3章介绍的向量空间模型将文档内容表示为特征向量,然后通过度量待分类文档的词项向量和其他代表文档或类别的向量的距离或夹角来确定该文档的类别。

本节将介绍两种空间分类算法:k近邻(k-nearest neighbor, kNN)和TF-IDF方法。每种方法都将待分类文档看成查询,通过从Lucene索引中搜索返回匹配的文档。而这些匹配文档的类别用于确定查询文档的类别。kNN算法在已分类的文档索引中搜索,而TF-IDF算法搜索的每篇索引文档代表一个目标类别。每种算法在实现和性能方面都有自己的优点。

向量空间模型是Lucene的核心,Lucene针对上述两种算法中所需要的距离快速计算进行了优化处理,从而为实现上述算法的功能提供了良好的基础。

本节将利用Apache Lucene来构建文档分类器,并使用kNN和TF-IDF算法将文档分到不同主题领域。你将利用免费获得的测试语料来训练这些分类器并学习如何对分类器的结果质量进行评估。作为一开始介绍的例子,我们会尽量使其保持简单,但是这里介绍的概念同样可以用于本章的其他例子当中。你也会注意到该例子的每一小节都对应第7.2节给出的每一个分类过程。

7.3.1 利用Lucene对文本进行分类

对于距离计算而言，Lucene是十分高效的工具。给定查询文档，即使在百万级文档的索引上Lucene也能在亚秒级响应并返回相似文档。Lucene返回的得分是两篇文档距离的倒数：Lucene返回的得分越高，那么两篇文档在向量空间中也越接近。每种算法中，与查询最接近的那些文档被用于确定查询的类别归属。

在kNN算法中，文档的类别基于其向量空间中邻居的类别来确定。kNN中的 k 是算法中的一个可调参数，它指的是确定类别时需要考察的邻居文档的数目。比如，如果 k 设为10，表示当确定一篇查询文档的类别时，需要考察其10个最近的邻居文档。

在TF-IDF算法中，对于每个目标类别创建单篇文档。在本节的例子中，每个类别文档只是该类别所有文档的累加结果。另一种做法是手工选择最有代表性的文档。这种方法之所以称为TF-IDF方法，是因为类别中词的权重计算采用了词项频率及逆文档频率，而这种权重计算方法构成了类别决策的基础。词项的相对重要性基于其出现的类别数目来计算。进一步而言，词项的重要性推动了查询词项的选择及查询文档与索引中类别的距离计算。kNN和TF-IDF方法的区别可以参考图7-3。

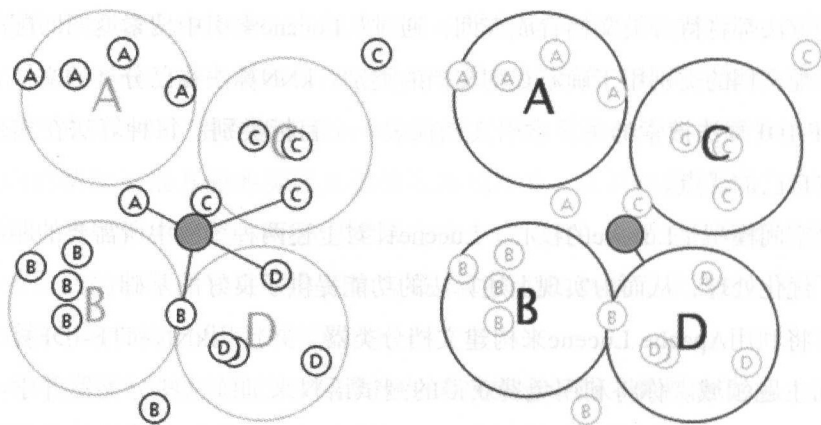


图7-3 kNN算法和TF-IDF算法的比较。左图中，待分类文档（灰色）与其最近的5个邻居（ $k=5$ ）相连。其中2个邻居属于类别C，而其他3个邻居分别属于类别A、B、D，因此，该文档会分配给类别C。右图中的大圆圈表示TF-IDF算法中使用的类别文档，它们是各自类别中所有文档的累加结果。根据右图的结果，待分类文档将会分配给离它最近的类别D而不是C

kNN和TF-IDF算法的实现可以共享大量代码。每种实现都基于训练数据构建Lucene索引。从Lucene API的角度来看,这意味着需要做两件事:第一,构建一个IndexWriter来配置对文本的分析过程;第二构建Document对象来索引。每个Document对象的内容因为分类算法的不同而不同。每个document至少有一个类别Field对象和一个内容Field。类别字段主要保存类别标签,而文档内容保存在内容字段。两个算法实现时共享的代码包括:读取并分析训练数据、将文档添加到索引中、对文档分类并评估算法的性能。

上述过程的一个重点是将待分类文档转换为Lucene查询。一种简单的做法是将文档的所有独立词项添加到查询中。对于短文档来说,这种做法可能已经足够,但是很快就会遇到长文档处理的难题。这类文档中的很多词可能对于类别决策没有作用。去除停用词尽管会减小基于文档的查询的大小,但是这样做有助于考虑索引的内容。搜索不在索引中出现的词项毫无意义,而搜索所有文档都出现的词项也毫无意义,但是此时会显著影响查询的时间。

为确定用于分类的最佳词,可以在索引构建过程中利用已经计算出的词项频率(TF)和文档频率(DF)。逆文档频率用于过滤掉那些重要性很低的词。最后会基于词项的相对重要性得到待分类文档中的查询词项列表。这样做就不会浪费时间来执行查询中一个对于确定文档到底属于A、B还是C类作用不大的词项。

幸运的是, Lucene开发者对于这些查询中的词项选择提供了十分方便的手段。Lucene中包含了一种称为MoreLikeThisQuery的查询类型,可以从查询文档中执行基于索引的词项选择过程。对于本节要构建的分类器来说,将使用MoreLikeThisQuery从输入数据中产生Lucene的Query对象。

本节将讨论基于Lucene的分类器实现,通过使用Lucene API来对文档切词、生成MoreLikeThis查询并对Lucene索引进行查询来获得输入文档的类别。在kNN算法的实现中, Lucene索引基于训练集中的每篇文档来构建,而在TF-IDF算法的实现中, Lucene基于类别文档来构建。由于两种算法共享了大量代码,它们会打成单个称为MoreLikeThis分类器的包,其中可以采用不同选项。该实现的代码在随书源代码的com.tamington.classifier.mlt包中。

7.3.2 为MoreLikeThis分类器准备训练数据

本例中将使用20个Newsgroup测试语料来训练MoreLikeThis分类器，从而将文档按照主题进行分类。该数据包含20个不同的互联网新闻组上张贴的文章，并划分成训练集和测试集。这些新闻组涵盖了很多主题领域，比如talk.politics.mideast和rec.autos这种区分很清楚的主题，也包括comp.sys.ibm.pc.hardware和com.sys.mac.hardware这种可能很相似的主题。训练数据中每个新闻组包含大概600篇文章，而在测试数据中每个新闻组包含的文章数接近400。由于每个目标类别的样本数目十分均匀，所以该数据集是一个很好的抽样集合。通过存档文件的名称就可以看出来，训练/测试集的划分基于时间来实现，其中训练数据由那些出现在测试数据之前的文件构成。

所用的20个Newsgroups测试语料可以从地址<http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz>下载。

下载并解压存档文件之后，会出现两个目录，分别是20news-bydate-train和20news-bydate-test。每个目录下包含多个子目录，每个子目录对应一个新闻组。而每个新闻组子目录下包含该新闻组的一个文件。为训练和测试分类器，必须将这些文件转换为恰当的格式。为简单起见，本章所有例子中都使用Mahout支持的输入格式。运行如下命令可以将训练和测试数据转换为所用的格式。

```
$MAHOUT_HOME/bin/mahout \  
  org.apache.mahout.classifier.bayes.PrepareTwentyNewsgroups \  
  -p 20news-bydate-train \  
  -o 20news-training-data \  
  -a org.apache.lucene.analysis.WhitespaceAnalyzer \  
  -c UTF-8  
$MAHOUT_HOME/bin/mahout \  
  org.apache.mahout.classifier.bayes.PrepareTwentyNewsgroups \  
  -p 20news-bydate-test \  
  -o 20news-test-data \  
  -a org.apache.lucene.analysis.WhitespaceAnalyzer \  
  -c UTF-8
```

注意 本章例子中会看到环境变量\$MAHOUT_HOME和\$TT_HOME的引用。MAHOUT_HOME指的是Mahout 0.6安装的基本目录，该目录下包含一个bin目录，而bin目录下包含一个名字为mahout的脚本程序。TT_HOME指的是包含本

书源代码的根目录。该目录下也包含一个bin目录，而这个bin目录下包含一个名字为tt的脚本程序。这两个脚本程序都包装了很多Java命令，这些命令设置了对应发布中运行Java类的合适环境。利用上述类似的环境变量可以建构自己的工作目录，这样就可以保证这些例子所产生的数据和Mahout拷贝及本书代码区分开来。

在本书出版之时，Mahout将要发布0.7版。该版本的贝叶斯分类器进行了重要修改，因此对于本书的例子要确保使用的是Mahout 0.6版。关注一下本书的作者论坛来更新样例代码以便与Mahout 0.7版及更高版本兼容。

正如在上面命令中看到的那样，这里使用了WhitespaceAnalyzer来对输入数据进行简单的切词处理。在后面的训练和测试过程中，数据还将利用Lucene的EnglishAnalyzer进行词干还原和停用词去除处理，因此此时除了空格切分之外不需要进行任何其他处理。其他分类器，比如Mahout的贝叶斯分类器会从数据预处理的词干还原和停用词去除过程中受益。

上述准备命令会建立一系列包含训练数据和测试数据的文件。每个类别会有一个文件，而每个文件有多行，每行会根据tab字符分成两列。第一列包含的是新闻组的名称，而第二列包含一个字符串，给出的是训练文档的内容，其中换行符和tab符都已经去掉。下面给出了训练集一些文件的内容摘录。每行包含消息头以及消息内容。

```
alt.atheism ... Alt.Atheism FAQ: Atheist Resources Summary: Books,
...
alt.atheism ... Re: There must be a creator! (Maybe) ...
alt.atheism ... Re: Americans and Evolution ...
...
comp.graphics ... CALL FOR PRESENTATIONS ...
comp.graphics ... Re: Text Recognition ...
comp.graphics ... Re: 16 million vs 65 ...
...
comp.os.ms-windows.misc ... CALL FOR PRESENTATIONS ...
comp.os.ms-windows.misc ... Re:color or Monochrome? ...
comp.os.ms-windows.misc ... Re: document of .RTF Organization: ...
```

现在训练和测试数据已经准备好，下面就可以训练MoreLikeThis分类器了。

7.3.3 训练MoreLikeThis分类器

MoreLikeThis分类器的训练过程可以通过命令行来实现，该命令使用了本书例子中的代码。下面的命令使用7.3.1节讨论的kNN生成一个模型：

```
$TT_HOME/bin/tt trainMlt \
-i 20news-training-data \
-o knn-index \
-ng 1 \
-type knn
```

该命令利用前一节准备好的训练数据构建Lucene索引。而构建面对TF-IDF算法的索引十分简单，只需将-type参数设置成tfidf即可。

下面开始考察上述命令后面的代码。清单7-1给出了构建索引所需要的代码，并为索引训练数据构建文本处理流水线：

清单7-1 构建Lucene索引

```
Directory directory
    = FSDirectory.open (new File (pathname)) ;
Analyzer analyzer
    = new EnglishAnalyzer (Version.LUCENE_36) ;
if (nGramSize > 1) {
    ShingleAnalyzerWrapper sw
        = new ShingleAnalyzerWrapper (analyzer,
            nGramSize, // min shingle size
            nGramSize, // max shingle size
            "-", // token separator
            true, // output unigrams
            true) ; // output unigrams if no shingles
    analyzer = sw;
}
IndexWriterConfig config
    = new IndexWriterConfig (Version.LUCENE_36, analyzer) ;
config.setOpenMode (OpenMode.CREATE) ;
IndexWriter writer = new IndexWriter (directory, config) ;
```

← ① 创建索引目录

← ② 建立分析器

← ③ 建立 shingle 过滤器

← ④ 创建 IndexWriter

在①处，一开始通过构建一个Lucene Directory对象来表示索引在磁盘上存储的位置。这里通过基于索引创建目录的完整路径调用FSDirectory.open ()方法来创建Directory。在②处要对分析器进行实例化，该分析器用于从文本输入中产生词条。

Lucene的EnglishAnalyzer是一个很好的起点，它能提供不错的词干还原及停用词去除功能。Lucene也提供了其他的分析器实现，或者也可以以外部库的方式来实现其他分析器。应该试试不同的选项来获得最适合你的应用的词条结果。例如，可能需要选择一个能够处理英文之外的语言的分析器，或者需要一个过滤器去掉字母数字词项或者对像*Wi-Fi*一样的词进行归一化表示。正如第3章介绍的那样，Solr的标准配置版本中有各种分析器及其组合的例子。

本例使用了 n 元组对Lucene EnglishAnalyzer的输出进行了增强处理。当nGramSize参数大于1时，Lucene ShingleAnalyzerWrapper可以用于在单独的一个个词之外产生 n 元组，❸处的代码给出了这一过程。

SHINGLE分析器及基于词的 n 元组 4章遇到 n 元组时，我们考察的是基于字符的 n 元组。而Lucene的shingle分析器产生的是基于词的 n 元组。当处理的是文本*now is the time*而nGramSize设为2时，shingle分析器将产生*now-is*、*is-the*和*the-time*这类词条（这里假设没有去掉停用词）。每个词条将作为特征来使用，它们可能在确定不同类别时有用。在上述具体情况下，ShingleFilter会对EnglishAnalyzer的输出结果进行处理。如果输入中的停用词*is*和*the*已经去掉，那么就会输出*now-*和*-time*这种 n 元组。其中的下划线给出的是去掉的停用词的位置，这样就可以避免给出不在原始文本中的词对。

本节一开始运行训练器的命令中使用了默认的nGramSize参数1。这可以通过在命令中增加一个-ng参数并在其后面增加一个数字来实现，比如-ng2。

当EnglishAnalyzer建立并可能与ShingleAnalyzerWrapper一起包装之后，就可以用于创建Lucene索引了。在❹处创建的IndexConfig和IndexWriter将用于构建训练数据的索引。

现在已经从磁盘的文件中读取了训练数据并将它们转换Lucene的Documents，然后将Documents添加到索引中。一开始必须要创建Field对象来保存每篇文档中的信息。

本例中有三个字段：文档的ID、文档的类别以及文档的内容，其中文档的内容字段包含了分析器产生的词条，这些词条将用作训练的特征。清单7-2给出了上述每个字段的创建过程。

清单7-2 建立文档字段

```
Field id = new Field ("id", "", Field.Store.YES,
    Field.Index.NOT_ANALYZED, Field.TermVector.NO) ;
Field categoryField = new Field ("category", "", Field.Store.YES,
    Field.Index.NOT_ANALYZED, Field.TermVector.NO) ;
Field contentField = new Field ("content", "", Field.Store.NO,
    Field.Index.ANALYZED, Field.TermVector.WITH_POSITIONS_OFFSETS) ;
```

对ID或类别字段不需要分析或构建词项向量。这些向量都存储在索引中用于后续检索。利用清单7-1中的分析器对内容字段进行分析，为记录每个词项在原始文档中的顺序，为该字段构建的词项向量包含完整的词项位置和偏移信息。

训练器代码在输入文件的每篇文档上进行循环，并根据所用分类算法来对文档构建索引。清单7-3给出了如何为kNN算法建立文档，此时每个训练样本都以单篇文档方式在索引中存在。

清单7-3 为kNN分类索引训练文档

```
while ((line = in.readLine ()) != null) {
    String[] parts = line.split ("t") ;           ← ① 收集内容
    if (parts.length != 2) continue;
    category = parts[0];

    categories.add (category) ;
    Document d = new Document () ;               ← ② 构建文档
    id.setValue (category + "-" + lineCount++) ;
    categoryField.setValue (category) ;
    contentField.setValue (parts[1]) ;
    d.add (id) ;
    d.add (categoryField) ;
    d.add (contentField) ;

    writer.addDocument (d) ;                     ← ③ 索引文档
}
```

在kNN算法的实现中，训练器首先在①处读取一个类别的每篇文档，然后在②处生成文档，最后这些文档在③处添加到Lucene的索引中。使用这种方法，索引的大小与训练集中的文档数目成正比。

清单7-4给出了如何为TF-IDF算法索引训练数据的过程。

清单7-4 为TF-IDF分类索引训练文档

```
StringBuilder content = new StringBuilder ();
String category = null;
while ((line = in.readLine ()) != null) {
    String[] parts = line.split ("t");
    if (parts.length != 2) continue;
    category = parts[0];
    categories.add (category);
    content.append (parts[1]) .append (" ");
    lineCount++;
}

in.close ();

Document d = new Document ();
id.setValue (category + "-" + lineCount);
categoryField.setValue (category);
contentField.setValue (content.toString ());
d.add (id);
d.add (categoryField);
d.add (contentField);

writer.addDocument (d);
```

← ① 收集内容

← ② 构建文档

← ③ 索引文档

在上述TF-IDF算法的实现中，训练器在①处读取一个类别的每篇文档然后将所有内容拼接成单个字符串。当某个类的所有文档读完之后，会在②处生成一篇Lucene文档，然后在③处将这篇文档添加到Lucene索引中。如果每个类别中的文本量很大的话，那么该算法消耗的内存会比kNN算法多，这是因为一个类别中的所有文档内容传递给Lucene之前，必须要在内存中构建一个缓冲区来保存这些内容。

现在已经为训练数据构建了索引，下面就可以实现并测试分类算法。

7.3.4 利用MoreLikeThis分类器对文档进行分类

对文档分类的第一步是打开Lucene的索引并建立分析器用于分析待分类的文本。比较重要的一点是，这里创建的分析器和训练中所用的分析器是一样的并且配置也一样，这样才能使得查询中的词项的生成方式和索引中词项的生成方式一模一样。这也意味着二者使用相同的停用词表、词干还原算法和 n 元组设置方式。当索引和

分析器预备好之后，就创建并配置MoreLikeThis类的一个实例。清单7-5给出了具体做法。

清单7-5 MoreLikeThis分类器构建

```
Directory directory = FSDirectory.open (new File (modelPath)) ;

IndexReader indexReader = IndexReader.open (directory) ;           ← ❶ 打开索引
IndexSearcher indexSearcher = new IndexSearcher (indexReader) ;

Analyzer analyzer                                                  ← ❷ 建交分析器
    = new EnglishAnalyzer (Version.LUCENE_36) ;

if (nGramSize > 1) {                                              ← ❸ 构建 n 元组
    analyzer = new ShingleAnalyzerWrapper (analyzer, nGramSize,
        nGramSize) ;
}

MoreLikeThis moreLikeThis = new MoreLikeThis (indexReader) ;      ← ❹ 建立 MoreLikeThis
moreLikeThis.setAnalyzer (analyzer) ;
moreLikeThis.setFieldNames (new String[] {
    "content"
});
```

在❶处创建Directory实例并打开IndexReader和IndexSearcher。IndexReader将用于返回构建查询的词条并在查询执行之后返回文档的内容。在❷处构建EnglishAnalyzer。如果设置启用的话，那么为了产生 n 元组，可以在❸处选择将EnglishAnalyzer包装在一个ShingleAnalyzer中。在❹处，创建一个MoreLikeThis类，给它传入一个IndexReader的实例，设置分析器，对它进行配置以使用内容字段的词条信息来选择在查询中使用的词条。当建立查询时，MoreLikeThis将考察查询文档及索引中词条的频率来确定使用哪些词条。那些在查询或索引中低于指定频率的词条，或那些在索引中频率过高的词条会候选在查询词条中去掉，这是因为它们对查询几乎不增加任何区分能力。

现在已经建立了构建和执行查询所需的对象，下面就可以执行搜索过程返回文档的类别。为了展示分类的执行过程，清单7-6详细给出了检索文档所用的方法。

清单7-6 利用MoreLikeThis对文本进行分类

```

Reader reader = new FileReader (inputPath) ;           ← ① 创建查询
Query query = moreLikeThis.like (reader) ;
TopDocs results
    = indexSearcher.search (query, maxResults) ;       ← ② 执行搜索
HashMap<String, CategoryHits> categoryHash
    = new HashMap<String, CategoryHits> () ;
for (ScoreDoc sd: results.scoreDocs) {                 ← ③ 收集结果
    Document d = indexReader.document (sd.doc) ;
    Fieldable f = d.getFieldable (categoryFieldName) ;
    String cat = f.stringValue () ;
    CategoryHits ch = categoryHash.get (cat) ;
    if (ch == null) {
        ch = new CategoryHits () ;
        ch.setLabel (cat) ;
        categoryHash.put (cat, ch) ;
    }
    ch.incrementScore (sd.score) ;
}
SortedSet<CategoryHits> sortedCats                      ← ④ 对类别排序
    = new TreeSet<CategoryHits> (CategoryHits.byScoreComparator ()) ;
sortedCats.addAll (categoryHash.values ()) ;
for (CategoryHits c: sortedCats) {                     ← ⑤ 展示类别
    System.out.println (
        c.getLabel () + "t" + c.getScore () ) ;
}

```

在①处创建一个Reader来读入待分类文档的内容。该内容会传递给MoreLikeThis.like () 方法，该方法完成的任务是基于文档中的关键词项来产生一条Lucene查询。现在有了查询，就可以在②处进行搜索并获得标准的Lucene应答，即一个包含匹配文档的TopDocs对象。在③处，在每个返回文档上执行循环，检索出其类别，然后收集类别的名称和得分到对象CategoryHits中。当在所有结果上循环迭代结束时，就在④处进行排序，即对CategoryHits对象集合进行排序然后在⑤处显示出来。排名最高的类别就是分配给文档的类别。这种评分和排名算法虽然比较原始，但能产生不错的结果。我们鼓励读者探索不同的类别评分方法并通过评估过程来确定其中的最佳方法。

不论索引是采用kNN还是TF-IDF算法来构建，类别的选择过程是一样的。在

kNN的情况下，对于每个类别可能有一篇多或篇文档，而对TF-IDF来说每个类别只有一篇文档。每篇文档的最终得分基于与查询匹配的文档的得分来计算。

将MoreLikeThis分类器集成到生产环境十分简单，只需要在应用程序的生命周期内执行一次设置操作，然后对每篇待分类文档，设置MoreLikeThis查询并作为搜索的一部分返回排好序的类别结果。

在本书随附代码的样例代码中，上面的每个任务都集成到一个称为MoreLikeThis的类中。该类可以作为生产环境下分类器部署的一个起点。在该类中会看到清单7-6所示的代码，尽管该代码在组织上与随书代码稍有不同，但是却执行的是相同的设置和分类操作。下一节将使用该类对分类器的精度进行评估。

7.3.5 测试MoreLikeThis分类器

利用下列命令来测试MoreLikeThis分类器。

```
$TT_HOME/bin/tt testMlt \  
-i category-mult-test-data \  
-m knn-index \  
-type knn \  
-contf content -catf category
```

当上述命令执行完毕时，将会给出两个指标来反映分类器输出结果的质量。第一个指标给出的是测试文档实例中分对和分错的百分比。在7.2.4节中我们称这个指标为精确率。第二个指标是一个称为混淆矩阵的表格，它给出了测试文档分类中的成功和失败情况。矩阵每列和每行代表的是算法可以为输入文档分配的类别。行代表测试文档事先属于的类别标签（即正确类别），而列代表的是分类器分配给文档的类别。矩阵中的每个元素都是一个数字，它代表的是预先分配给某个类别（对应行）的测试文档中被分类器分配给另一个类别（对应列）的数目。如果某个文档对应的行和列的类别相同则表示其被正确分类。下面给出了运行testMlt命令之后可能看到的完整混淆矩阵的一部分：

```

=====
Summary
-----
Correctly Classified Instances      :      5381    71.4418%
Incorrectly Classified Instances    :      2151    28.5582%
Total Classified Instances          :      7532

=====
Confusion Matrix
-----
a      b      c      d      e      f      ... <--Classified as
315  3      4      5      0      20    ... | 393  a    =rec.motorcycles
0     308  0      1      0      2      ... | 390  b    =comp.windows.x
0      0    320  4      1      0      ... | 372  c    = talk.politics.mideast
2      3     13    271  9      0      ... | 361  d    = talk.politics.guns
1      0     10     19   129  0      ... | 246  e    = talk.religion.misc
18     3      2      6      2    293    ... | 394  f    = rec.autos
...
Default Category: unknown: 20

```

混淆矩阵可以用于任意二类或多类分类器的评估。在二类分类器的情况下，混淆矩阵就是一个与表7-1类似的 2×2 的四元素矩阵。混淆矩阵总是 $N \times N$ 的方阵，其中 N 是分类器训练的目标类别数目。

在上例中，我们对混淆矩阵做了限制，只输出20个Newsgroups语料的前6个类别的分类结果情况。每行代表语料库中的一个类别，而每列会给出被分配给该类别的测试文档数目。本例中的类别a代表的是rec.motocycles。在本类别的393篇测试文档中，分类器正确地给其中的315篇赋予了rec.motocycles类。本行当中其余列的数字给出的是属于rec.motocycles的测试文档被赋予其他类别的情况。该矩阵表明其中的20篇测试文档被分配给rec.autos类。考虑到这两个新闻组主题领域和用词之间可能的相似度，出现上述结果不算意外。

我们会注意到，由于矩阵的对角线上的数字都是每行的最大值，所以该分类器倾向于将正确类别分配给测试文档。通过上述矩阵可以识别分类器可能出现问题的区域，比如有关motorcycle的20篇文档被分到auto类，而18篇auto类的文档又被分到motocycles类。此外，在多个talk类别之间看着也容易出现混淆，比如246篇talk.religion.misc的测试文档中只有129篇被正确分类，其中有19篇被分到talk.religion.guns类。在文档错误分类的情况下，混淆矩阵能够给出被错误分到的类别并显示训练数

据中的歧义之处。

清单7-7展示了通过读取训练数据中的一系列文件来测试一个训练好的MoreLikeThis分类器的过程。每篇文档被分类然后我们将分类结果和预先的类别进行比较。我们使用Apache Mahout中的ResultAnalyzer类来收集这些判定结果并给出前面介绍的指标。

清单7-7 对MoreLikeThisCategorizer的结果进行评估

```
final ClassifierResult UNKNOWN = new ClassifierResult ("unknown",
    1.0) ;
ResultAnalyzer resultAnalyzer =                                ← ① 创建 ResultAnalyzer
    new ResultAnalyzer (categorizer.getCategories (),
        UNKNOWN.getLabel () ) ;
for (File ff: inputFiles) {                                    ← ② 读取测试数据
    BufferedReader in =
        new BufferedReader (
            new InputStreamReader (
                new FileInputStream (ff),
                "UTF-8")) ;
    while ((line = in.readLine ()) != null) {
        String[] parts = line.split ("t") ;
        if (parts.length != 2) {
            continue;
        }
        CategoryHits[] hits                                    ← ③ 分类
            = categorizer.categorize (new StringReader (parts[1])) ;
        ClassifierResult result = hits.length > 0 ? hits[0] : UNKNOWN;
        resultAnalyzer.addInstance (parts[0], result) ;        ← ④ 收集结果
    }
    in.close () ;
}
System.out.println (resultAnalyzer.toString ()) ;              ← ⑤ 展示结果
```

在①处利用目标类别表以及一个无法分类情况下的默认UNKNOWN类来创建ResultAnalyzer。在②处从输入文件中将测试数据读到parts数组中。Parts[0]包含的是类别标签，parts[1]包含的是训练文档文本。文档在③处进行分类，得到待分类文档的一个类别排序表。然后，将排名最高的类别在④处添加到resultAnalyzer中。如果无法从分类器得到任何结果，则使用默认的UNKNOWN类。处理完所有训练数据之后，在⑤处展示正确的分类数目及混淆矩阵。

7.3.6 将MoreLikeThis投入生产环境

前面已经介绍了一个基于Lucene的文档分类器的基本构成模块，包括如何与必要的Lucene API进行交互，通过构建待分类文档的Lucene索引来训练分类器，如何通过将文档转换成Lucene查询来对它们进行分类，以及对分类的结果进行评估。同时也介绍了一些将这些算法用到生产环境下的基本知识。我们会在7.4.7节在某个部署场景下对这些知识进行扩展。当要将MoreLikeThis分类器部署到生产环境时，这里介绍的场景很容易改造以适应这个需求。Lucene API具备高度的灵活性，因此很容易将这类分类器集成到其他场景中。

Lucene的索引API使得分类模型的修改十分容易。对于kNN分类，对模型进行增强很容易，只需要增加更多的已分类文档到索引中即可。训练过程可以增量式进行，主要只受限于索引的规模。对于TF-IDF模型而言，只需要将已有的类别文档替换成新类别文档同时删除旧文档即可。这里分类器可以以增量方式增加新训练文档的能力称为在线学习，常常是分类算法所期望的性质之一。而基于离线学习算法的分类器不能采用这种方式来扩展，每次需要提高的时候，它们必须要从头开始训练，这样可能会消耗大量的时间和CPU。

现在已经对利用Lucene实现一个基于距离的分类方法有所了解，下面我们将在7.4节重复这个过程，这里使用Apache Mahout来训练一个朴素贝叶斯文本分类器。除了探讨这个基于统计的分类算法之外，我们还会考察如何将已有的数据（比如Web采集器的采集结果）进行改造以便能够用作训练数据。

7.4 利用Apache Mahout训练朴素贝叶斯分类器

第6章介绍了如何利用Apache Mahout对文档按照主题领域进行聚类。Mahout中也包含了多个分类算法，可以利用它们来对文本文档分配类别标签。其中的一个算法是朴素贝叶斯算法。该算法可以用于多种分类问题，也是了解概率分类方法的一个很好的入门。为了进行类别分配，概率分类算法会使用概率分类技术基于给定类别的文档特征出现概率来构建一个模型。

本节将使用Mahout中的朴素贝叶斯算法的实现来构建一个文档分类器。本章第一个例子中我们展示了如何利用20个Newsgroup测试语料来训练基于Lucene的分类

器。而这个例子将介绍如何利用从互联网上采集的数据来构建自己的测试语料并用于分类器训练。你将使用聚类那章采集的内容来构建训练和测试集。我们会展示训练分类器是怎样的一个迭代过程，并且给出为提高分类精确率而对训练数据进行重组的策略。最后，我们会展示如何将文档分类器集成到Solr中以便文档能够在索引时自动分配给相应类别。下面首先介绍朴素贝叶斯分类算法的理论基础。

7.4.1 利用朴素贝叶斯算法进行文本分类

朴素贝叶斯算法是一个概率分类算法。它利用训练数据中导出的概率来对输入文档进行类别分配的决策。训练过程分析训练文档中词和类别之间以及类别和整个训练集的关系。基于贝叶斯定理可以从训练集上生成词的集合（一篇文档）属于某个类别的概率。

这里算法的“朴素”一说主要来自词在给定类别中出现的独立性假设。直观来说我们知道某个给定主题领域词在文本中的出现不是独立的。像*fish*一样的词可能出现在包含*water*的文档的机会要比包含*space*的文档的机会要大。因此，朴素贝叶斯算法产生的概率不是真实概率。不过，只要是相对指标就有用。这些概率可能不能预测文档属于某个类别的绝对概率，但是通过比较词项*fish*属于每个类别的概率，它们可以用于确定包含*fish*的文档更可能与海洋而不是太空旅行有关。

训练时，朴素贝叶斯算法计算每个词出现在类别中文档中的次数并除以该类别中词的总出现次数。这称为条件概率，即某个词出现在某个特定类别的概率，这通常写作 $P(\text{Word}|\text{Category})$ 。假设对于类别Geometry来说只拥有一个包含3篇文档的很小训练集，其中词*angle*出现的次数是类中所有词出现次数的1/3，那么任意标为*geometry*的文档有0.33或33%的可能性包含词*angle*³。

可以利用每个单独的词概率然后将它们相乘来确定给定类别下文档的概率。严格来说，这本身并没有什么作用，但是贝叶斯定律提供了一个方法来将这些计算转换为给定文档下类别的概率，而这正是分类问题的本质所在。

贝叶斯定律指出，给定文档下类别的概率等于给定类别下文档的概率乘上类别的概率然后除以文档的概率。用公式表示为：

3 原文有误，这里做了修改。另外，本文给出的朴素贝叶斯算法基于多项式模型实现，另一种实现模型是贝努利模型。——译者注

$$P(\text{Category} \mid \text{Document}) = P(\text{Document} \mid \text{Category}) \times P(\text{Category}) / P(\text{Document})$$

上面已经介绍了如何计算给定类别下文档的概率。类别的概率等于训练文档中属于某个类别的训练样本数目除以训练文档的总个数。而在这里文档的概率可以不要，因为它只是一个放缩因子。如果将 $P(\text{Document})$ 设置为1，那么上述公式产生的结果在不同类别之间具有可比性。通过在每个目标类别上计算上述概率，就可以确定文档最可能属于的类别，只要每次计算中 $P(\text{Document})$ 都大于0，那么这些结果之间关系的相对顺序就是一样的。

上述解释提供了一个很好的起点，但是仍然只是管中窥豹。Mahout中朴素贝叶斯分类算法的实现包含大量的改进以应对在文本处理时算法失效的情况，比如前面提到的独立词项的问题。这些改进的一些描述参考Mahout wiki和Rennie等人发表的论文“Tackling the Poor Assumptions of naive Bayes Text Classifiers”（参考Rennie [2003]）。

7.4.2 准备训练数据

分类器只可能表现得和输入一样好。训练数据的数量、其组织的方式及选择的特征会作为训练过程的输入，并在分类器对新文档分类的精度当中起着重要作用。

本节介绍为Mahout贝叶斯分类器准备训练数据的过程。我们还会展示从Lucene索引中抽取数据的过程，并给出通过使用已有数据的属性来自举产生训练集的过程。在例子最后你会明白不同的自举方法如何影响分类器的最终质量。

第6章介绍了如何利用Solr建立一个简单的聚类应用。该应用从一系列RSS源中导入内容并将它们存储在Lucene索引中。本章将会利用该索引来构建训练集。如果还没有利用Clustering Solr示例来获取数据的话，那么现在就按照6.3节的指令在多天内运行Data Import Handler多次以获得恰当的训练文档集。在收集一些数据之后，可以考察一下索引以确定哪些数据可以用于训练。

现在在Lucene索引中有一些数据，接下来需要看看数据并确定如何利用它们训练分类器。有多种方法可以用于浏览Lucene索引中存储的数据，但是迄今为止最容易使用的是Luke。我们将考察一下数据以确定文档中哪些字段可以用作分类机制所需的类别源，我们还将确定一个用其内容进行训练类别集合，然后抽取出文档并将它

们写成训练数据格式输送给Mahout。贝叶斯分类器训练过程会分析具体类别中文档的词，并生成一个基于所包含的词确定文档类别的模型。

可以从地址<http://code.google.com/p/luke>下载Luke的最新版本文件lkeall-version.jar，其中version是Luke当前的版本号。下载JAR文件之后，运行命令`java -jar lukeall-version.jar`就可以启动Luke。

启动之后会遇到一个对话框，通过该窗口可以浏览文件系统以选择想要打开的索引。选择包含Lucene索引的目录并点击OK就可以打开索引（其他默认选项保留即可）。

当利用Luke浏览索引时，会注意到很多数据源提供了文档类别。这些类别的标签可能高度一般化，比如Sports，也可能十分具体，比如Baseball甚至New York Yankees。可以以这些信息为基础来组织训练数据。这里的目标是构建一系列词项列表用于将文章分到粗粒度的类别中从而对分类器进行训练。下面给出索引中名称为categoryFacet的字段的排名最高的12个类别，每个类别下有一些属于该类别的文档：

```
2081 Nation & World
923 Sports
398 Politics
356 Entertainment
295 sportsNews
158 MLB
128 Baseball
127 NFL
115 Movies
94 Sounders FC Blog
84 Medicine and Health
84 Golf
```

你会注意到，National & World类别中出现的是2081，这个代表类别中文档数目的数字会随着类别快速下降，比如在排名第12的类别Golf中，该数字仅仅为84。你还会注意到有一些主题交叉的领域，如Sports、Baseball和MLB，或者是相同主题不同的表达，如Sports和sportsNews。对数据进行清理以便能够高效用于训练是你自己的任务。训练数据的准备一定要注意，这一点很重要，这是因为其对分类器的精确率具有重要影响。为说明这一点，我们一开始将给出一个简单识别训练文档的方法，

然后给出一个更复杂的策略，最后比较它们的结果之间的差异。

从索引中找到的类别列表中，可以看到一些有用的词项出现在表的前几项。通过Luke发现的在索引中的一些其他类别也可以增加进来。

```
Nation
Sports
Politics
Entertainment
Movies
Internet
Music
Television
Arts
Business
Computer
Technology
```

用你最喜欢的文本编辑器，将上述内容保存在一个叫training-categories.txt的文件当中。现在你已经拥有了感兴趣的类别列表，利用类别列表和Lucene索引作为输入，运行extractTraningData工具：

```
$TT_HOME/bin/tt extractTrainingData \
--dir index \
--categories training-categories.txt \
--output category-bayes-data \
--category-fields categoryFacet, source \
--text-fields title, description \
--use-term-vectors
```

该命令将从Lucene索引中读取文档并在类别和源字段搜索匹配类别。当training-categories.txt中的某个类别在文档中找到，词项就会从title和description字段中存储的词项向量中抽取出来。这些词项将写为category-bayes-data目录下的一个文件。对每个类别而言都会有一个单独的文件。每个文件都是纯文本文件，可以采用任何文本编辑器或显示工具来阅读。

如果选择深入浏览这些文件，可能会注意到每一行对应Lucene索引中的单篇文档。每行都由基于tab字符分隔开的两列组成：类别名称出现在第一列，文档中出现的词项包含在第二列中。Mahout中的贝叶斯分类器期望输入字段已经进行词干还原

处理，因此你看测试数据中会看到这一点。extractTrainingData命令中的--use-term-vector参数会导致每篇文档词项向量中的词干还原词项被使用。

```
arts 6 a across design feast nut store world a browser can chosen ...
arts choic dealer it master old a a art auction current dealer ...
arts alan career comic dig his lay moor rest unearh up a a ...
business app bank citigroup data i iphon phone say store account ...
business 1 1500 500 cut job more plan tech unit 1 1500 2011 500 ...
business caus glee home new newhom sale up a against analyst ...
computer bug market sale what access address almost ani bug call ...
computer end forget mean web age crisi digit eras existenti face ...
computer mean medium onlin platon what 20 ad attract billion ...
```

当ExtractTrainingData类执行完之后，它会输出每个类别中发现的文档数目，显示出与如下类似的结果：

```
5417 sports
2162 nation
1777 politics
1735 technology
778 entertainment
611 business
241 arts
147 music
115 movies
80 computer
60 television
32 internet
```

注意到某些类别比其他类别中出现的文档数目要多。这可能影响分类器的精确率。某些像朴素贝叶斯一样的分类算法对非均衡数据较敏感，这是因为具有大量文档的类别当中的特征概率将比训练文档很少的类别的特征概率要更精确。

自举 (BOOTSTRAPPING) 利用简单规则构建训练文档集的过程称为自举。本例中利用了关键词来匹配已有文档类别名称的方法来对分类器完成自举过程。自举过程往往是必需的，这是因为正确标注的数据往往难以获取。很多情况下训练精确分类器的数据并不足够。其他情况下数据来自不同的数据源，这些数据源的分类机制并不一致。这种关键词自举的方法能够让你基于文档描述中常见

词的存在来对文档分组。给定类别中并非所有文档都满足这种特定规则，但是这足以让你产生足够的对分类器进行正确训练的样本。目前存在大量自举技术。有些技术涉及短文档的生成，这些短文档可以作为类别的种子，而有些技术利用其他算法的输出结果，比如可以利用上一章的聚类算法的结果或者其他类型的分类器的结果。自举技术常常会组合以使用额外数据进一步增强训练集。

7.4.3 留存测试数据

现在必须从产生的训练数据中保留一部分用于测试。在训练分类器之后，利用该模型来对测试数据分类并验证分类器的结果是否与已知结果相同。在本书附带的代码中，我们提供了一个称为SplitBayesInput的简单分割工具。我们会在解压任务写入的目录下点击SplitBayesInput，运行之后会生成两个额外目录：一个包含训练数据，另一个包含测试数据。可以利用下列命令运行SplitBayesInput。

```
$TT_HOME/bin/tt splitInput \
-i category-bayes-data \
-tr category-training-data \
-te category-test-data \
-sp 10 -c UTF-8
```

这种情况下，我们取每个类别中文档的10%写到测试目录下，其余文档则写到训练数据目录下。SplitBayesInput类提供了多种不同的方法来选择训练集/测试集分割方法。

7.4.4 训练分类器

当使用SplitBayesInput准备好训练数据之后，就该着手准备训练第一个分类器了。如果运行在Hadoop集群上，将训练和测试数据复制到Hadoop的分布式文件系统下，运行如下命令来构建分类模型。如果没有运行在Hadoop集群下，那么数据就可以不管-source hdfs参数而直接从当前的工作目录下读取。

```
$MAHOUT_HOME/bin/mahout trainclassifier \
-i category-training-data \
-o category-bayes-model \
-type bayes -ng 1 -source hdfs
```

训练节时间却取决于训练数据的规模，也取决于训练过程是在本地还是在Hadoop集群下分布式运行。

当训练成功完成之后，结果模型会写入到命令中指定的输出目录。模型目录下包含一系列Hadoop SequenceFile格式的文件。Hadoop SequenceFile包括键/值对，其通常是使用Hadoop MapReduce框架进程的输出结果。键和值可以是原生类型或者Hadoop序列化的Java对象。Apache Mahout自带了很多用于观察这些文件内容的工具。

```
$MAHOUT_HOME/bin/mahout seqdumper \  
-s category-bayes-model/trainer-tfIdf/trainer-tfIdf/part-00000 | less
```

Trainer-tfidf目录包含了朴素贝叶斯算法用于分类的所有特征列表。一旦导出就是下面这个样子。

```
no HADOOP_CONF_DIR or HADOOP_HOME set, running locally  
Input Path: category-bayes-model/trainer-tfIdf/trainer-tfIdf/part-00000  
Key class: class org.apache.mahout.common.StringTuple  
Value Class: class org.apache.hadoop.io.DoubleWritable  
Key: [__WT, arts, 000]: Value: 0.9278920383255315  
Key: [__WT, arts, 1]: Value: 2.4908377174081773  
...  
Key: [__WT, arts, 97]: Value: 0.8524586871132804  
Key: [__WT, arts, a]: Value: 9.251850977219403  
Key: [__WT, arts, about]: Value: 4.324291341340667  
...  
Key: [__WT, business, beef]: Value: 0.5541230386115379  
Key: [__WT, business, been]: Value: 7.833436391647611  
Key: [__WT, business, beer]: Value: 0.6470763007419856  
...  
Key: [__WT, computer, design]: Value: 0.9422458820512981  
Key: [__WT, computer, desktop]: Value: 1.1081452859525993  
Key: [__WT, computer, destruct]: Value: 0.48045301391820133  
Key: [__WT, computer, develop]: Value: 1.1518455320100698  
...
```

对文件进行探查往往很有用，这样就能确定训练中的特征是否真正与抽取特征有关联。探查该结果可以告诉你可能你并没有进行正确的停用词去除处理，也可能词干还原工具有点问题，或者没有产生预期的 n 元组结果。探查训练特征的数目也很

重要，这是因为特征集的规模会影响Mahout贝叶斯分类器的内存消耗。

7.4.5 测试分类器

当分类器训练完成之后，就可以利用前面提到的留存数据作为测试数据对分类器进行评估。下列命令会将训练阶段得到的模型加载到内存并对测试集中的每篇文档进行分类。分类器分配给每篇文档的标签将与人工给定的标签进行对比，所有文档的结果将会计算出来：

```
$MAHOUT_HOME/bin/mahout testclassifier \
  -d category-test-data \
  -m category-bayes-model \
  -type bayes -source hdfs -ng 1 -method sequential
```

当测试过程完成之后，会给出两种评估指标：一个是分类精确率，另一个是混淆矩阵。这些指标在7.3.5节已经介绍过。

```
=====Summary
-----
Correctly Classified Instances      :      906      73.6585%
Incorrectly Classified Instances    :      324      26.3415%
Total Classified Instances          :      1230
=====
Confusion Matrix
-----
a   b   c   d   e   f   g   h   i   j   k   l   <--Classified as
0   0   0   0   5   0   0   0   1   0   3   2   |  11  a  = movies
0   0   0   0   0   0   0   0   1   0   1   4   |  6   b  = computer
0   0   0   0   0   0   0   0   0   0   1   2   |  3   c  = internet
0   0   0   4   0   0   0   5   4   0   4   42  |  59  d  = business
0   0   0   1   26  0   0   6   10  0   18  10  |  71  e  = enter...
0   0   0   0   2   0   0   0   1   0   3   0   |  6   f  = television
0   0   0   0   7   0   1   0   0   2   4   0   |  14  g  = music
0   0   0   0   0   0   0   103  43  0   10  10  |  166 h = politics
0   0   0   0   1   0   0   25   145  0   16  10  |  197 i = nation
0   0   0   0   8   0   0   3   7   1   3   1   |  23  j  = arts
1   0   0   0   1   0   0   1   7   0   493  4   |  507 k = sports
0   0   0   0   0   0   0   15   12  0   7   133  |  167 l = technology
Default Category: unknown: 12
```

这种情况下，可以使用混淆矩阵来调整你的自举过程。上述矩阵表明，分类器

在对体育类文档进行分类处理时效果不错，507篇体育相关的文档中有493篇被分到这个类别中。而技术类效果也同样不错，该类别中的167篇文档中有133篇被正确分类。而视频类结果就不太好，该类别所有11篇文档中没有一篇被分到该类。视频类文档分到的最多的类别是娱乐类。考虑到视频是娱乐的一种形式，并且训练集中娱乐类文档的数目（71篇）显著多于视频类（11篇），出现上述结果也算合理。这也展示了非均衡训练集和重叠样本所带来的结果。由于娱乐类训练文档数目显著高于视频类，娱乐类明显压过了视频类，同时也可以看到娱乐类分类发生错误的内容与国家、体育和技术类相关，只是因为这些类别的训练文档数目比娱乐类更多的原因。该实例表明，如果主题领域分开得更好、训练集更均衡，那么可以获得更好的精确率。

7.4.6 改进自举过程

在前面的例子中，我们使用了单个词项来定义每个类别的文档。通过寻找所有在数据源或类别字段包含类别词项的文档，ExtractTrainingData会对每个类别构建一组文档。这样做会产生一个混淆多个类别的分类器，其原因是分配给每个类别的训练集存在主题相似性和非均衡性。为解决这个问题，可以使用一组相关词项来定义每类文档。这会让你将所有与体育相关的类别合并成单个体育类别，而所有娱乐类类别合并成另一个类别。为了合并相似的类别，这种方法也能允许你在Lucene索引中进一步达到文档池并返回额外的训练样本。

创建一个叫做 training-categories-mult.txt 包含下列标签的文件。

```
Sports MLB NFL MBA Golf Football Basketball Baseball
Politics
Entertainment Movies Music Television
Arts Theater Books
Business
Technology Internet Computer Science
Health
Travel
```

在上述文件中，每行的第一个词变成了类别的名字。一行的每个词都用于搜索文档。如果行中的任意一个词匹配上了文档类别或源字段的词项，那么该文档会写入该类别对应的训练数据文件。例如，任意类别字段包含字符串MLB的文档将会看

成体育类的一部分，而类别字段包含词项*music*的文档将会是娱乐类，而类别字段包含*Computer*的文档将归入技术类。

使用如下命令再次运行ExtractTrainingData。

```
$TT_HOME/bin/tt extractTrainingData \
--dir index \
--categories training-categories-mult.txt \
--output category-mult-bayes-data \
--category-fields categoryFacet, source \
--text-fields title, description \
--use-term-vectors
```

输出结果将会写入categories-mult-bayes-data目录，并且在终端显示下列文档数目。

```
Category document counts:
5139 sports
1757 technology
1676 politics
988 entertainment
591 business
300 arts
173 health
12 travel
```

根据上述训练样本的数目，很可能不能训练出一个能够精确判定旅游类的分类器，因此可能要考虑收集额外的训练文档或者此时干脆完全去掉旅游类，但是这里为了结果显示的需要还是保留了该类。

再次执行分隔、训练和测试过程如下。

```
$TT_HOME/bin/tt splitInput \
-i category-mult-bayes-data \
-tr category-mult-training-data \
-te category-mult-test-data \
-sp 10 -c UTF-8
$MAHOUT_HOME/bin/mahout trainclassifier \
-i category-mult-training-data \
-o category-mult-bayes-model \
-type bayes -source hdfs -ng 1
$MAHOUT_HOME/bin/mahout testclassifier \
-d category-mult-test-data \
```

```
-m category-mult-bayes-model \
-type bayes -source hdfs -ng 1 \
-method sequential
```

测试阶段的输出表明生成了一个改进的分类器，该分类器能有79.5%的机会进行正确分类：

Summary

```
-----
Correctly Classified Instances      :      846    79.5113%
Incorrectly Classified Instances    :      218    20.4887%
Total Classified Instances          :      1064
Confusion Matrix
-----
```

a	b	c	d	e	f	g	h	<--Classified as	
0	0	0	0	0	0	1	0	1	a = travel
0	3	0	0	8	0	5	43	59	b = business
0	0	2	1	7	1	2	4	17	c = health
0	1	0	57	12	1	19	9	99	d = entertainment
0	0	0	0	142	0	14	12	168	e = politics
0	0	0	17	3	3	4	3	30	f = arts
0	1	0	3	9	0	495	6	514	g = sports
0	1	0	1	23	0	7	144	176	h = technology

Default Category: unknown: 8

从上述结果会看到，分类器的输出结果已经提高了6%，该数字应该还不错。尽管刚才的做法朝着正确的方向进行，但从混淆矩阵来看很明显需要解决其他一些问题。

幸运的是，对于本例的目标来说，获取训练数据和选择分类机制已经具有显著的灵活性。首先，很明显，由于大部分旅游类文档来说都没有正确分类，因此对于旅游类来说没有足够的训练数据。而健康和艺术类面临同样的问题，它们当中大部分的文档也都被分错。大部分艺术类文档被分到娱乐类这一事实也意味着两个类值得合并。

7.4.7 将Mahout贝叶斯分类器集成到Solr

分类器训练之后，必须要部署到生产环境。本节将展示如何将Mahout贝叶斯分类器以文档分类器身份集成到Solr搜索引擎的索引过程。当Solr将数据加载到Lucene

索引时,同时也运行文档分类器产生类别字段的值,该值可以用作附加的搜索词项或者用于结果的多面展示。

通过建立一个定制的Solr UpdateRequestProcessor来实现上述过程,在索引收到更新请求时会调用上述对象。该对象初始化时,会加载训练好的Mahout贝叶斯分类器,然后对每篇文档进行处理、分析和分类。UpdateProcessor会将类别标签以SolrDocument字段的方式加入并添加到Lucene索引中。

首先通过在solrconfig.xml中定义,在Solr加入一个定制的更新需求处理链(参考org.apache.solr.update.processor.UpdateRequestProcessorChain)。该链定义了多个用于创建更新过程所需对象的工厂类。BayesUpdateRequestProcessorFactory将产生用于处理每次更新的对象并分配一个类别,RunUpdateProcessorFactory将处理更新并将它添加到Solr构建的Lucene索引中,而LogUpdateProcessorFactory会追踪更新的统计信息并将它们写入到Solr日志中。整个过程如清单7-8所示。

清单7-8 更新请求处理链在solrconfig.xml中的配置

```
<updateRequestProcessorChain key="mahout" default="true">
  <processor class=
    "com.tamingtext.classifier.BayesUpdateRequestProcessorFactory">
    <str name="inputField">details</str>
    <str name="outputField">subject</str>
    <str name="model">src/test/resources/classifier/bayes-model</str>
  </processor>
  <processor class="solr.RunUpdateProcessorFactory"/>
  <processor class="solr.LogUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

可以使用inputField、outputField和model参数配置BayesUpdateRequestProcessorFactory,其中inputField参数给出的是包含待分类文档的字段名,outputField参数给出的是写入类标签的字段名,model参数给出的是分类所用模型的路径。而defaultCategory参数是可选的,如果指定,那么定义的就是当分类器无法确定文档类别时所赋予的类别名称。当输入文档不包含模型中的任何特征时,这种情况常会发生。当Solr启动并初始化其插件时,该工厂类就会创建。此时就会验证这些参数并通过MahoutDatastor对象的初始化来加载模型。分类算法会建立,并且ClassifierContext对象会使用这些元素中的每一个进行初始化。

清单7-9给出了分类模型如何加载到InMemoryBayesDatastore的过程。

清单7-9 建立Mahout ClassifierContext

```
BayesParameters p = new BayesParameters ();
p.set ("basePath", modelDir.getCanonicalPath ());
Datastore ds = new InMemoryBayesDatastore (p);
Algorithm a = new BayesAlgorithm ();
ClassifierContext ctx = new ClassifierContext (a, ds);
ctx.initialize ();
```

上述方法对于中等规模的特征训练出的小模型来说还行，但是对于无法放入内存的模型来说不太现实。Mahout给出了另一种datastore来从HBase获取数据。这种方法的实现和前面一样都足够直接简单。

当ClassifierContext初始化之后，它会作为BayesUpdateRequestProcessorFactory成员变量而保存，并当Solr收到更新请求注入到每个新的实例化BayesUpdateRequestProcessor中。每个更新请求以一个或多个SolrInputDocuments的格式到达。Solr API使得从文档中抽取一个字段十分简单，这样就可以使用前面初始化的分类器上下文来对文档进行预处理和分类。清单7-10给出了利用Solr分析器进行预处理的过程，该分析器基于Solr schema中input Field的配置执行正当的预处理操作，然后结果写入到一个String[]数组，Mahout分类器上下文会以该数组作为输入。Solr分类器遵循Lucene 分析器的API，因此这里给出的切词代码可以用于任何使用Lucene分析器的上下文。

清单7-10 使用Solr分析器对SolrInputDocument切词

```
String input = (String) field.getValue ();
ArrayList<String> tokenList = new ArrayList<String> ();
TokenStream ts = analyzer.tokenStream (inputField,
    new StringReader (input));
while (ts.incrementToken ()) {
    tokenList.add (ts.getAttribute (CharTermAttribute.class).toString ());
}
String[] tokens = tokenList.toArray (new String[tokenList.size ()]);
```

当得到分类器可以处理的词条之后，获得结果是一件非常简单的事情，只须调用Mahout ClassifierContext的classifyDocument方法即可。清单7-11给出了该过程如何返

回一个包含文档分配类别标签的ClassifierResult对象的过程。当无法定义类别时，比如输入文档和模型没有任何公共词时，该classify方法也会使用一个默认类别。在获得标签之后，只要ClassifierResult不是NULL或者等于defaultCategory参数值时（这种情况下表示为NO_LABEL），其将作为一个新字段分配给SolrInputDocument。

清单7-11 利用ClassifierContext来对SolrInputDocument分类

```
SolrInputField field = doc.getField (inputField) ;
String[] tokens = tokenizeField (inputField, field) ;
ClassifierResult result = ctx.classifyDocument (tokens,
    defaultCategory) ;
if (result != null && result.getLabel () != NO_LABEL) {
    doc.addField (outputField, result.getLabel ()) ;
}
```

这种方法的一个缺点是，为了分类需要在内存中保存切词的结果，当然这取决于索引文档的类型。将来Mahout或许会扩展为直接利用词条流作为输入。上述方法的第二个缺点是在索引时对文档字段进行两次高效切词处理。第一次在分类时进行，而第二次在处理流时进行，后一次是为了将词条添加到Lucene索引中。

除上述缺点之外，上述方法提供了一个十分有效的机制，该机制可以在文档加入到Solr索引时对文档进行分类，也能展示如何使用Mahout贝叶斯分类器的API来通过编程方式对文档进行分类。其中一种或者两种机制都可以用于自己的项目，在这些项目中可以在文档索引时对文档进行标注或者使用Mahout分类器对文档进行分类。

本节中主要探讨了朴素贝叶斯分类算法，该算法是一种统计分类算法，它通过来自训练数据集的观察结果来确定给定类别下某个词集合的概率。然后，该算法利用贝叶斯定律来转换条件概率来得到给定文档词集合下的类别概率。下一节中将给出另一个统计分类算法，该算法也对给定词集合下的类别建模，但它不需要确定其相关的关系概率。

本节也考察了如何从Web上收集训练数据的技术。我们考察了自举的过程，实验了多种对训练文档集分割的办法并展示了训练数据对分类器精确率的影响。7.5节中我们会继续上述探索过程，其中我们会引入命名实体的使用来增强训练数据以提高最后的结果。

7.5 利用OpenNLP进行文档分类

7.4节介绍的朴素贝叶斯分类算法是一种概率算法，它基于训练数据中特征和类别的关系来进行分类。本节将利用另一个统计算法——OpenNLP中的最大熵算法来进行文本分类。最大熵算法用于构建模型的信息与贝叶斯算法相同，但是在构建模型时采用了不同的方法。MaxEnt算法使用了一个回归算法来确定文档特征和类别的关系。其训练过程反复分析训练语料来寻找每个特征的权重并生成一个数学公式，该公式将产生与训练数据中观察到的结果最相似的结果。本节将对回归模型进行一个基本的回顾，这样就可以解释其运行原理并将这些概念与文本分类的核心任务关联起来。

本节当中会介绍的OpenNLP的另一个有用部分是命名实体发现API。在第5章介绍命名实体（人物、位置等）识别的时候曾经介绍过这个API。本节将利用这些识别出的实体来提高MaxEnt文档分类器的性能。除了将训练数据中的每个独立词看作特征之外，OpenNLP确定的多个词组合成的命名实体（如*New York City*）也将作为特征使用。

OpenNLP命名实体识别工具也是个分类器。它训练成可以基于多种不同特征来识别命名实体词。OpenNLP自带了用于抽取一系列命名实体类型的模型，因此为了利用API不必训练自己的命名实体识别器，尽管需要时也可以选择这样做。

因此，除了使用OpenNLP来对文档进行分类之外，要使用OpenNLP的一个独立部分来生成分类决策所需的特征。这是所谓背页式处理的一个例子，这里某个分类器（文档分类器）利用另一个分类器（命名实体识别器）的结果进行训练。这也是一种司空见惯的做法。你可能也碰到过确定句子边界、词边界或词性时需要分类器生成分类特征的情况。你必须要注意这个事实，即接收数据的分类器性能与产生特征的分类器性能紧密相关。

本节最后，你将会理解最大熵分类器的工作过程，同时利用其代码中所使用的术语及其与文档分类的关系。我们给出的例子展示了OpenNLP文档分类API和命名实体识别API的工作过程，并带领你遍历从训练到结果质量评估的分类器构建过程。

7.5.1 回归模型及最大熵文档分类

OpenNLP MaxEnt分类器中使用的多项式logistic回归模型是多种回归模型中的一种。一般而言，回归模型与确定一个因变量和多个自变量的关系有关。每个回归模型都是一个特征加权的数学函数。当每个特征的值及其权重组合在一起然后进行特征加权，其结果代表模型的预测或输出。回归算法涉及基于训练集的观察结果来确定合适的特征权重。

图7-4给出了一个简单的用于预测计算机程序速度的回归模型。该模型将计算机中CPU的数目和程序运行的时间关联起来，模型中只有一个自变量或者说特征，即CPU的数目。因变量是处理时间。图中每个点代表训练集中的一个元素，给出的是具有某个特定数目CPU的计算机的程序运行时间。这里总共有5个训练样本，其中对于1个CPU是1000ms而对于8个CPU大约是100ms。

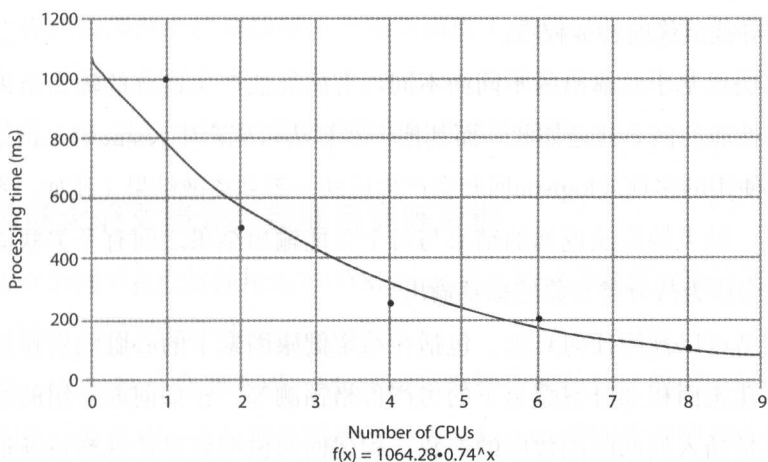


图7-4 一个简单的二维回归模型。每个点代表一个观测数据。回归函数给出的曲线出现在图下方区域，它能够对未观测数据进行预测。相同的原理可以用分类，其中每个词代表独立的一维

回归算法试图建立一个函数来预测拥有某个具体数量CPU的计算机运行程序的时间，而该CPU的数量在训练集中没有出现过。图中的曲线代表回归算法输出的结果函数。该结果表明，在一个3CPU的计算机上运行该程序大概需要450ms的时间，而在7CPU的机器上大概需要110ms。本例中回归算法生成的公式出现在图下方。这里将0.74作为CPU数量的幂然后乘以1064.28得到最后的结果。这里的0.74用于对自变量加

权，称为参数，而另一个值为1064.28的变量称为校正常数。

在回归模型中，每个特征都有一个权重参数，整个公式只有一个校正常数。回归算法会确定最优的权重参数和校正常数以得到一条和训练数据偏离最小的曲线。在这个简单的例子中，只有一个自变量，但是回归模型典型的使用情况是拥有大量自变量，每个自变量都有一个需要确定的参数。

存在多种形式的回归模型，它们组合自变量、参数和校正常数的方式有所不同。在前面的指数模型中，每个参数会作为其自变量的幂，最后每个自变量上的结果和校正常数相乘。在线性模型中，每个参数会和其自变量相乘，得到的结果求和并加上校正常数。每个回归公式会产生不同形状的结果，可能是条直线、曲线或更复杂的形状。

除了公式的基本结构之外，根据参数求解的不同回归算法也有所不同。这里你会遇到诸如梯度下降或迭代缩放之类的技术，为得到预期结果，每种技术采用了不同的方法来寻找最优的特征权重。

回归算法也基于其输出的不同而不同。有些算法产生连续的输出结果，如前面介绍的软件性能的例子就是如此。而其他一些算法可能产生yes/no的二值结果。最大熵分类器所使用的多项式logistic回归会产生反映一系列离散结果（比如一系列类别）的输出结果。最大熵算法返回的结果与每个可能输出结果之间有个关联概率，排名最高的结果会作为待分类对象的标签输出。

回归模型可以预测任何对象，包括在给定健康因素下的心脏病发作概率预测到给定位置、建筑面积和卧室数目下的房产价格预测等。正如前面介绍的那样，模型本身无非就是插入到回归函数中的参数。构建回归模型就是让观察特征拟合训练数据的输出结果，而使用模型产生结果实际就是填上特征值并利用模型中存储的权重产生最终结果。

上面介绍的这些内容到底如何与前面讨论文本分类时用到的概念发生关联？回归模型的输出（因变量）对应分类器的输出，即分类器产生的类别标签。回归模型的输入即自变量就是特征，而在文本分类上下文中就是文本的某个方面（如词项）。用于文本分类的回归模型会很大，这是因为语料中的独立词都会看成独立的自变量。由于文本的稀疏性，训练也更复杂一些，每个训练样本只有构成回归函数的相对很少的自变量相关的信息。可以设想，确定合适的特征权重对很多自变量而

言本身就足够困难，更不用说完整数据集无法提供的情况。

本章OpenNLP API和源码中所使用的文本分类术语比这里用于文本分类的语言更通用。为理解OpenNLP的文档分类，首先必须探索这些领域之间的关系。在OpenNLP分类器的命名体系当中，训练特征称为*predicate*。而这些*predicate*会带上下文出现，任意*predicate*可能出现在多种上下文中。在文档分类中，*predicate*是词或其他文档特征，而上下文则是文档。训练语料库由多个上下文构成。每个上下文与一个输出结果相关联。该输出结果等于分类器所分配的类别标签。这些概率也可以映射到回归模型的语言中。每个*predicate*（特征或词项）就是一个自变量。每个自变量用于预测因变量的值，即输出结果或类别标签。训练语料库由上下文构成，每个上下文将*predicate*映射为输出结果。这些映射是训练和测试所基于的观察结果。当训练模型时，通过比较观察结果和模型的结果来确定如何改进模型。

训练过程比较语料中找到的每一个独立词项（*predicate*）与所有词项产生的结果，并执行一系列迭代过程来寻找每个*predicate*的最优权重，以产生期望的输出结果。每次迭代会提高回归函数生成训练数据所示结果的能力。

7.5.2 为最大熵文档分类器准备训练数据

对于本章的例子我们将使用为7.4节Mahout 贝叶斯分类器所收集的训练数据，但是你也可以使用7.3节的20个Newsgroups数据。

与Mahout贝叶斯分类器不用，MaxEnt分类器会执行自己的词干还原过程来处理文本。因此，要使用一个稍微不同的命令来从Lucene索引中抽取训练数据。不同于从Lucene词项向量中抽取还原后的词项，下列命令将从索引中抽取每个字段的原始文本。

```
$TT_HOME/bin/tt extractTrainingData \  
  --dir index \  
  --categories training-categories.txt \  
  --output category-maxent-data \  
  --category-fields category, source \  
  --text-fields title, description
```

如果看看训练数据，就会看到词项没有进行词干还原，词仍然以混合大小写方

式存在。这里大小写很重要，因为它是MaxEnt分类器寻找命名实体的重要线索之一。

```
arts    6 Stores Across the World Are a Feast for Design Nuts A few ...
arts    For Old Masters, It's Dealers' Choice While auction houses ...
arts    Alan Moore Digs Up 'Unearthing' and Lays His Comics Career ...
...
business Citigroup says iPhone banking app stored data Citigroup ...
business United Tech plans 1,500 more job cuts HARTFORD, Conn. - ...
business New-home sales up, but no cause for glee New-home sales ...
...
computer What's for Sale on the Bug Market? Almost any ...
computer The Web Means the End of Forgetting The digital age ...
computer The Medium: What 'Platonic' Means Online Craigslist ...
```

下面使用splitInput将数据分割成独立的训练和测试集：

```
$TT_HOME/bin/tt splitInput \
-i category-maxent-data \
-tr category-maxent-training-data \
-te category-maxent-test-data \
-sp 10 -c UTF-8
```

现在已经准备好了训练和测试数据，下面训练最大熵文档分类器。

7.5.3 训练最大熵文档分类器

OpenNLP项目在发布版中提供了一个文档分类器，但是要使用它需要相当的编程量。下一节将介绍对基于OpenNLP DocumentCategorizer类构建的分类器进行训练和测试所需要的代码。

TestMaxent和TrainMaxent类基于OpenNLP实现了一个可以从命令行运行的分类器。trainMaxent命令用于训练分类器。-i参数指定的输入目录必须包含和前面例子一样格式的训练数据，即每个类别一个文件，每个文件的每行对应一个文档。MaxEnt文档分类器期望接收用空格分开并且没有事先进行词干还原或大小写归一化的文本，之所以保留大小写，是因为在进行命名实体识别时需要考虑大小写。-o参数用于指定MaxEnt模型写入的文件：

```
$TT_HOME/bin/tt trainMaxent \
-i category-maxent-training-data \
-o maxent-model
```

接下来深入考察训练OpenNLP文档分类模型所需的代码，并了解针对自己的目标对训练过程进行定制的方方面面。

为了训练MaxEnt模型，必须要建立输入目录和输入文件，创建原始数据源和将训练数据转换成特征的特征生成器，并将这些传输给训练器，训练器基于这些信息来建立训练集的统计模型。清单7-12展示了这一过程。

清单7-12 训练DocumentCategorizer

```
File[] inputFiles = FileUtil.buildFileList (new File (source)) ;
File modelFile = new File (destination) ;
Tokenizer tokenizer = SimpleTokenizer.INSTANCE;           ← ① 创建数据流
CategoryDataStream ds =
    new CategoryDataStream (inputFiles, tokenizer) ;
int cutoff = 5;
int iterations = 100;
NameFinderFeatureGenerator nffg                          ← ② 建立特征生成器
    = new NameFinderFeatureGenerator () ;
BagOfWordsFeatureGenerator bowfg
    = new BagOfWordsFeatureGenerator () ;

DoccatModel model = DocumentCategorizerME.train ("en",
    ds, cutoff, iterations, nffg, bowfg) ;                 ← ③ 训练分类器
model.serialize (new FileOutputStream (modelFile)) ;
```

在①处建立SimpleTokenizer和CategoryDataStream以从训练数据文件中抽取类别标签和词条。

在②处创建NameFinderFeatureGenerator和BagOfWordsFeatureGenerator类，这些类用于生成特征，这些特征包括文档中的原始词项以及OpenNLP命名实体发现工具识别的命名实体。

当数据流和特征生成器构建之后，利用DocumentCategorizerME来训练分类模型。在③处首先将数据流、特征生成器和训练参数传输给train () 方法然后将训练得到的模型序列化到磁盘上。

为了理解训练数据如何转换为分类器训练所用的格式，值得近距离观察一下切

词和特征生成的过程。清单7-13给出了使用CategoryDataStream来从训练数据中生成DocumentSample的过程。

清单7-13 从训练数据中生成DocumentSample

```
public DocumentSample read () {
    if (line == null && !hasNext ()) {           ← ❶ 读取训练数据行
        return null;
    }
    int split = line.indexOf ('t') ;              ← ❷ 抽取类别
    if (split < 0)
        throw new RuntimeException ("Invalid line in "
            + inputFiles[inputFilesIndex]) ;
    String category = line.substring (0, split) ;
    String document = line.substring (split+1) ;
    line = null; // mark line as consumed
    String[] tokens = tokenizer.tokenize (document) ;   ← ❸ 切词
    return new DocumentSample (category, tokens) ;      ← ❹ 构建样本
}
```

在❶处，CategoryDataStream的read () 方法通过调用hasNext () 方法从输入数据中获得一行行的数据。hasNext () 隐式读取训练数据的新行并将它们保存在line变量中，当到达训练数据的结尾时，将line置为null。在读取训练数据的每一行时，❷处的代码抽取类别和文档数据。然后文档数据在❸处进行切词处理来产生一个用于训练过程所用特征的词项集合。最后，在❹处利用类别标签和训练样本中的词条来构建一个DocumentSample对象。

在DocumentCategorizerME中，DocumentSample集合通过DocumentCategorizer-EventStream传递给特征生成器。这会产生模型训练所需的结果。这些结果由输出结果和上下文组成。其中，输出结果为类别标签，而上下文是对文档内容切词之后产生的词集。

CategoryDataStream创建的DocumentSample事件对象通过NameFinderFeature-Generator和BagOfWordsFeatureGenerator处理成特征。BagOfWordsFeatureGenerator作为OpenNLP API的一部分被提供，它会返回以特征集合方式表示的文档样本中的词条。NameFinderFeatureGenerator使用了OpenNLP的NameFinder API来寻找词条中的命名实体并将它们以特征方式返回。OpenNLP NameFinder和为寻找命名实体加载的多个模型封装在NameFinderFactory中，其负责的就是寻找并加载用于识别命名实体

对象的多个模型。清单7-14给出了NameFinderEngine寻找并加载用于识别命名实体的模型的过程。

清单7-14 加载命名实体发现模型

```
File modelFile;

File[] models                                ← ❶ 寻找模型
    = findNameFinderModels (language, modelDirectory) ;
modelNameNames = new String[models.length];
finders = new NameFinderME[models.length];
for (int fi = 0; fi < models.length; fi++) {
    modelFile = models[fi];
    modelNameNames[fi] = modelNameFromFile (language, modelFile) ;
    log.info ("Loading model {}", modelFile) ;
    InputStream modelStream = new FileInputStream (modelFile) ;
    TokenNameFinderModel model =
        new PooledTokenNameFinderModel (modelStream) ;
    finders[fi] = new NameFinderME (model) ;
}
```

❷ 确定模型的名称

❸ 读取模型

在❶处findNameFinderModels () 方法扫描模型目录以加载模型文件。然后每个模型文件加载到NameFinderFactory维护的数组中。每个模型加载时，在❷处会使用modelNameFromFile () 通过去除任意前导路径和末尾后缀将模型文件名转换为模型名称。在❸处PooledTokenNameFinderModel会执行模型读取、解压以及将结果写入内存的繁重工作。每个模型加载时，会利用加载的模型创建NameFinderME类的实例。这些模型中的每一个都存储在NameFinderFactory.getNameFinders () 方法返回的一个数组中。

现在已经加载了NameFinder类的实例，该实例可用于识别输入中的命名实体。清单7-15的代码来自NameFinderFeatureGenerator类，可以用于对CategoryDataStream返回的DocumentSamples执行命名实体识别操作。

清单 7-15 利用NameFinderFeatureGenerator生成特征

```
public Collection extractFeatures (String[] text) {
    NameFinderME[] finders = factory.getNameFinders () ;
    String[] modelNameNames = factory.getModelNames () ;

    Collection<String> features = new ArrayList<String> () ;
```

❶ 获得命名实体识别器

```

StringBuilder builder = new StringBuilder ();
Listing 7.14 Loading nam
for (int i=0; i < finders.length; i++) {
    Span[] spans = finders[i].find (text) ;
    String model = modelNames[i];
    for (int j=0; j < spans.length; j++) {
        int start = spans[j].getStart () ;
        int end = spans[j].getEnd () ;
        builder.setLength (0) ;
        builder.append (model) .append ("=") ;
        for (int k = start; k < end; k++ ) {
            builder.append (text[k]) .append ('_') ;
        }
        builder.setLength (builder.length () -1) ;
        features.add (builder.toString ()) ;
    }
}
return features;
}

```

← ② 识别命名实体

← ③ 抽取命名实体

← ④ 收集命名实体

在①处获得NameFinderME的引用及NameFinder工厂类加载的模型名称。NameFinderFactory在平行的两个数组中存储了命名实体识别器及其对应的名称。每个模型将用于识别不同的命名实体类型，比如地名、人名、时间和日期等。

在②处利用引擎加载的每个NameFinderME通过调用find方法对输入词条进行处理。该方法返回的数组中的每个span会给出引用点，这些引用点会利用起始和结束偏移来给出代表原始文本中命名实体的一个或多个词条。在③处利用这些偏移位置来生成将要存储为特征的字符串。每种情况下，模型名称会预先安排在字符串前面从而得到诸如location=New_York_City的特征。

所有生成的特征会收集到一个列表中，该列表会在④处返回给文档分类器。

运行训练过程会得到与下面类似的结果：

```

Indexing events using cutoff of 5
Computing event counts... done. 10526 events
Indexing...
done.
Sorting and merging events... done. Reduced 10523 events to 9616.
Done indexing.
Incorporating indexed data for training...

```

```
done.
Number of Event Tokens: 9616
    Number of Outcomes: 12
    Number of Predicates: 11233
...done.
Computing model parameters...
Performing 100 iterations.
  1: .. loglikelihood=-26148.6726757207   0.0024707782951629764
  2: .. loglikelihood=-24970.114236056226  0.6394564287750641
  3: .. loglikelihood=-23914.53191047887   0.6485793024802813
...
99: .. loglikelihood=-7724.766531988901   0.8826380309797586
100: .. loglikelihood=-7683.407561473442   0.8833982704551934
```

在训练之前，文档分类器必须将训练中要用的特征组织在索引中，这样就可以在训练过程中快速访问这些特征。上面前几行输出结果给出了该过程的结果。每篇已标注训练文档都会产生一个事件，而在索引过程中重复事件会被计数，某些文档在不包含有用特征情况下可能会被剔除。

训练器输出中的截断值指的是TF的截断值。训练语料中任意出现次数少于5的词条会被忽略。任意只由次数少于截断值的词条组成的文档也会被剔除。Predicate代表用于训练的词条，这里会看到语料库中总共有11233个独立predicate。这些predicate包括OpenNLP BagOfWordsFeatureGenerator产生的单词条以及NameFinderFeatureGenerator产生的命名实体。在回归模型中，每个predicate都是一个自变量。

上述输出也表明，当索引过程结束时，在去重之后总共有12种输出结果和9616个训练样本。

当索引过程结束时，可以开始观察训练过程的输出结果。该过程自己包含100次迭代，每次迭代代表对整个训练数据集合的一遍扫描，每次扫描都是为了调整模型参数并确定回归函数的输出。对于每次迭代，模型的输出结果会与观察输出结果进行比较从而计算出对数似然率。

对数似然是度量两个模型相似度的一个指标。不要将它用作一个绝对指标，而是用作每次迭代之间模型变化的一个相对指标。你可能期望每次迭代后模型的结果趋近于零，这也意味着模型每一步都十分接近训练数据中的观察结果。如果对数似然远离零的话，意味着模型越来越差，训练数据中可能存在问题。你也会注意到在最初的

几步训练中对数似然的变化会比后面的过程更显著。如果注意到对数似然在迭代100次之后仍然持续显著变化的话，那么就只可能使用更多的迭代来执行训练过程。

当达到100次迭代之后，训练器会将模型写入到磁盘，之后你就拥有了一个文档分类模型。下一节给出了一些在测试过程中使用模型来分类所需的API调用。

7.5.4 测试最大熵文档分类器

本节使用7.3.5和7.4.5节所用的相同方法来测试最大熵文档分类器，即对一系列已标注文档进行分类并将结果与原始标签进行比较。实现该过程的TestMaxent类可以使用如下命令来调用：

```
$TT_HOME/bin/tt testMaxent \
    -i category-maxent-test-data \
    -m maxent-model
```

这里使用的是extractTrainingData工具产生的测试数据以及trainMaxent命令产生的模型。当测试完成时，就会呈现我们所熟悉的精确率和混淆矩阵。

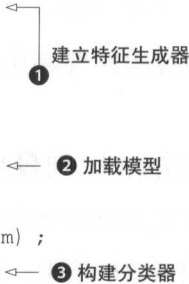
TestMaxent类展示了如何将已训练的模型加载到内存并用于对文档进行分类的过程。清单7-16的代码从磁盘加载模型并为文档处理准备好切词流水线。你会发现清单7-16的大部分代码都与清单7-12中训练分类器所使用的代码类似。

清单7-16 准备DocumentCategorizer

```
NameFinderFeatureGenerator nffg
    = new NameFinderFeatureGenerator ();
BagOfWordsFeatureGenerator bowfg
    = new BagOfWordsFeatureGenerator ();

InputStream modelStream =
    new FileInputStream (modelFile);
DoccatModel model = new DoccatModel (modelStream);
DocumentCategorizer categorizer
    = new DocumentCategorizerME (model, nffg, bowfg);
Tokenizer tokenizer = SimpleTokenizer.INSTANCE;

int catCount = categorizer.getNumberOfCategories ();
Collection<String> categories
    = new ArrayList<String> (catCount);
for (int i=0; i < catCount; i++) {
```



① 建立特征生成器

② 加载模型

③ 构建分类器


```

        categories.add (categorizer.getCategory (i)) ;
    }
    ResultAnalyzer resultAnalyzer =
        new ResultAnalyzer (categories, "unknown") ;
    runTest (inputFiles, categorizer, tokenizer, resultAnalyzer) ;

```

测试过程

准备结果分析器

一开始还是在①处构建特征生成器，然后使用DoccatModel类从磁盘加载模型②。之后该模型用于在③处创建DocumentCategorizer的一个实例。最后，在④处通过模型从分类器中获取类别列表来对ResultAnalyzer进行初始化。在⑤处运行测试过程。

在下一节中，我们将考察把最大熵文档分类器集成到生产环境下的代码。

7.5.5 生产环境下的最大熵文档分类器

现在已经加载了模型，建立了切词器、特征生成器、分类器及结果评估器，准备好对一些文档进行分类。清单7-17给出了如何对从文件中读入的文档进行处理、分类并把结果传送给ResultAnalyzer来评估7.3.5节的MoreLikeThis和7.4.5节的贝叶斯分类器。

清单7-17 利用DocumentCategorizer对文本进行分类

```

for (File ff: inputFiles) {
    BufferedReader in = new BufferedReader (new FileReader (ff)) ;
    while ((line = in.readLine ()) != null) {
        String[] parts = line.split ("t") ;
        if (parts.length != 2) continue;

        String docText = parts[1];
        String[] tokens = tokenizer.tokenize (docText) ;
        double[] probs = categorizer.categorize (tokens) ;
        String label = categorizer.getBestCategory (probs) ;
        int bestIndex = categorizer.getIndex (label) ;
        double score = probs[bestIndex];

        ClassifierResult result
            = new ClassifierResult (label, score) ;
        resultAnalyzer.addInstance (parts[0], result) ;
    }
    in.close () ;
}

System.err.println (resultAnalyzer.toString ()) ;

```

① 对文本进行预处理

② 分类

③ 分析结果

④ 展示结果

你应该还记得，每篇测试文档出现在输入文件单行的第二列中。一开始从每篇训练文档中抽取文本然后使用SimpleTokenizer产生一系列词条①。然后这些词条在②处传送给分类器。categorize方法利用清单7-16构建的BagOfWordsFeatureGenerator和NameFinderFeatureGenerator来生成特征，并利用模型来组合这些特征产生可能的输出结果，每种输出结果都带有模型计算推导出来的一个概率。每个输出结果对应于某个具体的文档类别，最后排名最高的类别最终分配给文档。在③处构建一个ClassifierResult并将之输送给ResultAnalyzer。当以上述方式处理完所有文档之后，在④处可以打印最后的结果概要。

将OpenNLP DocumentCategorizer集成到生产系统中所需的代码和本节介绍的代码之间的差异并没有那么显著。对于生产系统来说，必须要和清单7-16所示的那样只进行切词器、特征生成器和分类器的一次建立过程，而和清单7-17所用的类似代码进行文档分类。实际实现时考虑可能会对7.4.7节的例子进行调整，在那里利用的是OpenNLP文档分类器将Mahout贝叶斯分类器集成到Solr中。

现在我们已经给出了一系列可以基于内容对文档分类的算法，下面将探讨这些算法的一个应用，即内容标注，并探讨如何利用这些算法的变形通过提供基于主题的大型文档集浏览机制或搜索结果的主题面来组织文档内容。

7.6 利用Apache Solr构建标签推荐系统

在介绍一个自动内容标注器的实现之前，首先介绍一些背景知识，这包括标注如何成为内容定位的流行机制及其重要的原因。

在互联网的早期，出现了寻找内容的两种主要模式。搜索引擎对Web进行索引并提供简单搜索界面供用户输入用于内容定位的搜索关键词。其他网站将Web网页分类成主题类别树形式的大型目录结构。每一种方法从事的是一种不同的信息搜寻行为，都有自己的优缺点。

搜索引擎维护的索引结构和界面能够满足很多人的需要，但是当无法用关键词描述自己的信息需求时最终用户就会受到困扰。关键词搜索还会受限于这一事实而进一步复杂化：很多概念可以用很多不同词项来描述，因此如果搜索中使用的词项和文档中的词项无法关联的话，那么重要的内容可能会被丢失。因此，关键词搜索往往部分包括正确词项的猜测过程，也包括为得到额外查询词项对搜索结果的探索

过程。

其他网站开发了大型的组织机制并将Web页面分配到这些机制的类别当中。类别常常组织成树结构，离根节点更远的每一层节点意义上都更加具体。低层的宽泛类别（如艺术或旅游类）会让位于高层具体类别（如歌舞伎或东京旅游）。这些称为层次分类体系的机制被单个实体所管理，并往往成长到最终用户难以理解的地步。如何沿着分类体系浏览才能定位到有关日本烹饪的网页？Web成长的爆炸性使得分类体系无法跟上它的脚步。每个新网站必须要彻底精通分类体系的用户手工分类。信息搜寻者希望能够快速发现所要的内容而不是通过浏览一棵复杂的分类体系树来寻找答案。

社会化标注作为大型中心化组织管理的分类体系的替代物已经出现。与将内容强行组织成单个层次类别机制不同的是，社会化标注将内容的组织能力交给了用户。例如，当用户想要记住某个网页时，他们会用他们所理解的方式分配一些词给这个网页，这些词称为标签。Twitter用户会在推文中通过在关键词前面添加哈希标识符#来嵌入哈希标签，这样其他对此主题感兴趣的用户就能定位这些推文。

上述两种情况下，所使用的标签都是公开的，因此对相同标签感兴趣的任意用户都能找到该内容。由于上述标注行为被数十到数千用户所重复，因此基于公共词项的使用可以形成一个分类机制。每个标签的定义形成于其使用方式。与将内容强行组织成严格受控的分类体系不同，社会化内容标注利用大量用户的视角来定义内容的上下文。

这种有机形成的组织机制往往称作大众分类法，它可以看成是社会化、协同生成的分类体系的变形。大众分类法是组织或对对象进行分类的一种方式，该方式自然形成于大量用户随时间推移组织对象的方式。利用这种组织机制，内容不一定要与一个严格的分类体系相吻合，而是表示为一系列词项集合，每个词项从其自身角度都可以看成是一种类别。

在Web上标注内容的行为足够简单：你在Web上创建或者寻找一些感兴趣的事物，然后在浏览器中点击一个按钮，将当前事物与一些有助于你后来找到该事物的词关联起来。当标注了更多内容，将这些标签聚合在一起便可以看到十分有趣的结果。看看图7-5给出的标签云，该标签云给出了一个用户在delicious.com上给网页分配的所有标签的情况，难道你不能快速从中了解该用户的兴趣吗？

你可以在每个例子上利用Lucene构建一个标签推荐引擎。在7.3节介绍的基于距离的方法中，每篇文档或每个类别向量都使用单个类别来标注，并可以从多个匹配的候选类别中选择单个最相关的类别。在贝叶斯的那个例子中，你见证了对于每篇文档往往有很多好的类别这一事实，并且这些类别往往从一般性类别到具体或者是描述不同意义的不同方面的类别。与建立受限类别集合不同的是，如何才能利用文档集已有标签来生成其他文档的标签？

本节当中将实现这一点。下面我们会展示如何使用kNN分类算法，以及已经标注过的文档集合基于Apache Solr来构建一个标签推荐系统。和7.3节的kNN算法实现相似，你会构建一个包含训练文档的索引然后利用MoreLikeThis查询在索引中匹配文档。推荐的标签会从每个查询的结果中获得。

7.6.1 为标签推荐收集训练数据

为构建标签推荐系统，下面将使用问答网站Stack Overflow (<http://www.stackoverflow.com>) 上的一份数据集。Stack Overflow是一个“面向专业和狂热编程爱好者的协同编辑问答网站”。它由Stack Exchange公司运行，该公司运行面向多个不同主题领域的类似问答网站。用户访问每个网站并参与提问题、给出新答案或对已有用户社区的现存答案进行评级这些过程。提问者为每个问题分配了一系列关键词标签。到2011年1月，Stack Overflow导出数据包含超过400万个帖子。这些帖子中，大约有100万个帖子是带有标签的问题。这些问题的长度可能不同，但是大部分帖子提供了文本和标签，这些数据对于训练一个与马上要构建的标签推荐系统类似的系统十分有用。

除了作为一个优秀的训练数据源之外，Stack Overflow和其他姐妹网站也提供了一个例子，其面向内容网站的标签使用得很好。图7-7给出了描述java标签的网页。该页面给出了用词项java来标注的最流行的那些问题，它同时也提供该标签的一些统计信息，比如利用java标注的问题数量以及相关的标签及其数量。特别有用的一点是上面也给出了标签的定义。这可以防止用户在面对有关印度尼西亚或饮料的问题时误用该标签。

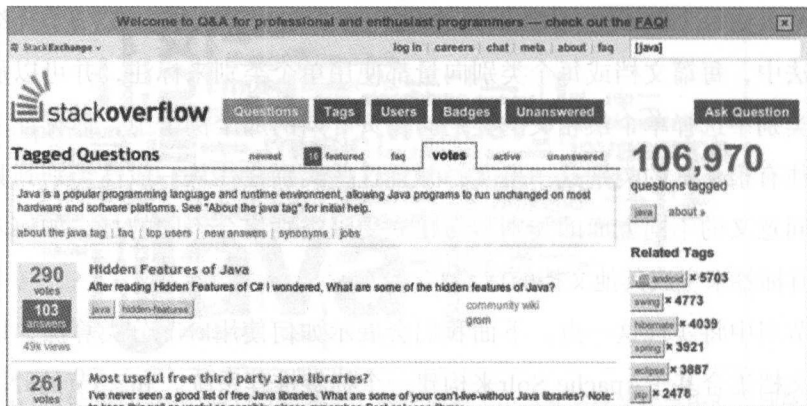


图7-7 stackoverflow.com上的一个网页，给出了用java作为标签的问题。该页面展示了一种浏览信息的方法，也给出了一种获得有用的训练数据的方法

下面从一个小规模的例子开始，数据集由一个10 000个帖子的训练集和1000个帖子的测试集组成。判断推荐标签是否正确的准则首先会稍微放宽一些。每个问题可能会有多个标签，因此可以将每个问题的文本看成相关联标签的训练集。如果给定问题包含标签`php`、`javascript`和`ajax`，那么就可以将该问题看成每个标签类别的训练样本。

当使用测试数据评估标注引擎的结果质量时，考察给定对象引擎分配的标签并将它们和用户分配的标签进行对比。如果看到一个或多个匹配的标签，可以认为存在一个匹配。

同本章开始探讨的例子类似，标签推荐系统可以通过Lucene索引训练文档来进行训练。下面将Solr当成一个平台，能够通过Data Import Handler快速加载和索引Stack Overflow数据，并将标签推荐系统的结果展示当成一个Web服务。

分类和推荐

术语分类和推荐中的每一个都正式描述了相关的机器学习算法族。两者的区别在于，分类从受控的选项列表中提供了一个很小的可能选项集合，而推荐提供的是一个更大的选项集合子集，这些选项来自几乎无穷的选项集合，比如产品目录或学术论文数据库。

除了或不同于基于内容的决策之外，很多推荐系统分析用户的行为来确定推荐目标，例如，跟踪你的喜好或者观察你所选择或购买的商品，并将这些信息与其他用户的行为进行比较，这样可以将他们喜欢的商品推荐给你。像

Amazon或Netflix一样的网站会使用这类推荐算法来向你推荐要买的图书或者要看的电影。

本例当中仅仅使用输入文档的内容来推荐标签，并且这些文章的内容已经在系统中存在。从这种方式上看与我们迄今讨论过的分类或归类算法更相似，而与基于行为的推荐系统相反。如果将两种方法结合会十分有趣。

如果对学习到更多的推荐系统知识感兴趣的话，请参考Manning出版社出的另一本书，即Owen、Dunning和Friedman著的*Mahout in Action*（Owen 2010）。

7.6.2 准备训练数据

开始本例的介绍之前，可以从地址<http://www.tamingtext.com>下载Stack Overflow的训练和测试数据，或者根据<http://blog.stackoverflow.com/category/cc-wiki-dump/>的说明直接从Stack Overflow网站下载全部数据集。从该页面可以下载一个torrent文件，然后就可以用你最喜欢的BitTorrent客户端来下载数据。数据导出的torrent文件中包含所有的Stack Exchange网站的文件，因此使用BitTorrent客户端只选择Stack Overflow的导出数据，该数据在一个名为Content\Stack Overflow 11-2010.7z的7-zip格式的文件中。

对上述归档文件解压之后，会得到badges.xml、comments.xml、posthistory.xml和posts.xml文件。下面会使用posts.xml作为训练数据源。该文件包含一系列嵌套在一个包含所查找数据的posts元素中的行元素。我们将简单介绍一下如何产生感兴趣的分割然后讨论文件自身的格式。

可以利用如下命令将上述文件分割成训练集和测试集：

```
$TT_HOME/bin/tt extractStackOverflow \  
-i posts.xml \  
-te stackoverflow-test-posts.xml \  
-tr stackoverflow-training-posts.xml
```

默认情况下，extractStackOverflow将抽取找到的前面100 000个问题作为训练文档，前面的10 000个问题作为测试集。你可以自由抽取更多或更少的数据来实验，以观察数据量如何影响训练的时间和质量。

在posts.xml文件中，有许多名字为row的XML元素。我们对寻找那些有标签的问

题的行感兴趣。每个问题出现在一行中，但是Stack Overflow数据中并非所有的行都是一个问题。extractStackOverflow工具会通过考察每行的PostType属性来过滤掉那些非问题行。该属性值为1的行是问题，将会被保留，而其他将被过滤掉。其他我们感兴趣的属性包括Title、Tags和Body。我们将使用这些属性作为原始训练数据。有些其他属性可能也用得着，因此我们也将它们保留。

问题的标签出现在每行中的tags属性中，其中多个标签通过<and>符号隔开。对个一条帖子而言，整个属性值可能看起来像<javascript><c++><multithreaded programming>这样，也就是说该帖子通过三个独立的标签*javascript*、*c++*和*multithreaded programming*来标注。当利用上述数据来训练和测试时必须要对该格式进行解析。

7.6.3 训练Solr标签推荐系统

我们将利用Stack Overflow数据和Solr数据导入handler来训练Solr标签推荐系统。下面提供了一个Solr实例的代码样例，该实例配置成读取训练数据文件、从XML中抽取必要的字段并将标签转换为离散值存储在索引中。部分配置信息可以参考清单7-18。

清单7-18 从dih-stackoverflow-config.xml中摘录的一部分

```
<entity name="post"
  processor="XPathEntityProcessor"
  forEach="/posts/row"
  url="../../../stackoverflow-corpus/training-data.xml"
  transformer="DateFormatTransformer, HTMLStripTransformer,
    com.tamingtext.tagrecommender.StackOverflowTagTransformer">
  <field column="id" xpath="/posts/row/@Id"/>
  <field column="title" xpath="/posts/row/@Title"/>
  <field column="body" xpath="/posts/row/@Body"
    stripHTML="true"/>
  <field column="tags" xpath="/posts/row/@Tags"/>
```

数据导入handler将使用XPathEntity处理器来将训练数据分开成多个独立的Lucene文档，其中每个文档对应<posts>标签内的每个<row>标签。<row>标签内的多个属性将用于填充索引中的ID、title、body和tags字段。Body属性的内容会剥离

HTML代码。

StackOverflowTagTransformer是一个简单的定制转换器，它显式搜索tags属性然后按照刚才介绍的方式来处理找到的内容。这样就为solr索引中的每篇文档产生多个独立的tag字段实例。清单7-19给出了该类的全貌。

清单7-19 在Solr数据导入handler实例中转换数据

```
public class StackOverflowTagTransformer {  
    public Object transformRow (Map<String, Object> row) {  
        List<String> tags = (List<String>) row.get ("tags");  
        if (tags != null) {  
            Collection<String> outputTags =  
                StackOverflowStream.parseTags (tags);  
            row.put ("tags", outputTags);  
        }  
        return row;  
    }  
}
```

数据导入handler每次将一行原始数据传递到transformRow方法中，转换器可以很自由地以多种方式来修改该行的每列。这个例子中，该行的所有列替换为一个新的标签集合，该集合从原始格式中解析。可以将row.get ("tags")调用的值当成List，这是因为对于本实例tags字段在Solr schema中定义为多值。

你会注意到如果考察全部的dih-stackoverflow-config.xml文件，那么数据导入handler配置也会从Stack Overflow数据中加入多个其他字段。

现在你对索引过程如何工作已经有所了解，下面可以通过运行Solr实例并将文档加载到该实例开始。通过运行如下命令来启动Solr服务器。

```
$TT_HOME/bin/start-solr.sh solr-tagging
```

Solr将会启动并在加载每个模块和配置项的同时将大量日志信息显示到终端上。几秒钟之后启动过程会结束，你会看到如下消息。

```
Started SocketConnector@0.0.0.0:8983
```

在终端上保留日志信息的显示，后面为了解决数据加载中出现的问题可能会用到这些信息。

现在已经成功启动了Solr，你必须要对数据导入handler的配置进行编辑以引用将用作训练数据的文件。编辑\$TT_HOME/apache-solr/solr-tagging/conf/dih-stackoverflow.properties文件并将URL的值从/path/to/stackoverflow-training-posts.xml转变为你系统中训练数据的完整路径。尽管property的命名为url，但一个常规文件路径也可以。修改该设置之后，保存dih-stackoverflow.properties文件并关闭编辑器。

访问http://localhost:8983/solr/admin/dataimport.jsp页面，点击指向DIH-SSTACKOVERFLOW的链接，将会进入Solr数据导入handler的开发主控屏幕。由于在启动Solr时已经编辑了数据导入handler的配置，因此必须要在你的浏览器中点击页面左边框架底部的Reload-config按钮来重新加载修改后的配置。

当配置重新加载成功之后，就准备好加载训练数据了。点击Reload-config按钮旁边的Full-import按钮，Solr会在加载训练数据的同时继续工作。可以通过点击浏览器右边框架中的Status按钮来以XML格式显示当前的状态。几分钟后，就会看到如下的状态信息。

```
Indexing completed. Added/Updated: 100000 documents. Deleted 0 documents
```

你也会在开始提交启动命令的终端中看到如下Solr中的日志输出信息：

```
Mar 11, 2011 8:52:39 PM org.apache.solr.update.processor.LogUpdateProcessor
INFO: {add=[4, 6, 8, 9, 11, 13, 14, 16, ... (100000 adds) ], optimize=} 05
```

现在已经完成了训练阶段，Solr实例已经准备好产生推荐标签了。

7.6.4 构建推荐标签

Solr-tagging 实例已经为使用Solr的MoreLikeThisHandler响应查询而配置。同前面给出的MoreLikeThis分类器一样，这里的查询handler将把文档作为输入并利用索引识别查询中最有用的词项来返回匹配文档。清单7-20给出了MoreLikeThisHandler在solrconfig.xml中的配置情况。

清单7-20 在solrconfig.xml中配置MoreLikeThisHandler

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler">
  <lst name="defaults">
    <str name="mlt.fl">title, body</str>
    <int name="mlt.mindf">3</int>
```

```
</1st>
</requestHandler>
```

下面使用的标签推荐方法类似于7.3.6节介绍的kNN分类算法。与kNN中对MoreLikeThisQuery返回的文档所分配的类别计数不同的是，这里对标签计数，然后利用这些标签来进行推荐。TagRecommenderClient负责将输入文档传送给Solr并为了对标签进行聚合、评分和排序而对结果进行后处理。清单7-21在更高层面上描述了上述过程。

清单7-21 利用TagRecommenderClient来生成推荐标签

```
public TagRecommenderClient (String solrUrl)
    throws MalformedURLException {
    server = new HttpSolrServer (solrUrl) ;
}

public ScoreTag[] getTags (String content, int maxTags)
    throws SolrServerException {
    ModifiableSolrParams query = new ModifiableSolrParams () ;
    query.set ("fq", "postTypeId:1")
        .set ("start", 0)
        .set ("rows", 10)
        .set ("fl", "*", "score")
        .set ("mlt.interestingTerms", "details") ;
    MoreLikeThisRequest request
        = new MoreLikeThisRequest (query, content) ;
    QueryResponse response = request.process (server) ;
    SolrDocumentList documents = response.getResults () ;
    ScoreTag[] rankedTags = rankTags (documents, maxTags) ;
    return rankedTags;
}
```

← ❶ Solr 客户端

← ❷ 查询参数

← ❸ 建立并执行请求

← ❹ 收集并对标签排序

首先必须建立到Solr的连接。在❶处，建立一个HttpSolrServer的实例，该实例不管名称叫什么，实际上都是使用HTTP客户端库的Solr客户端，它发送请求给Solr服务器。Solr服务器的URL作为参数提供。

建立客户端之后，必须要建立查询来返回与该查询相匹配的文档。你将使用这些返回文档的标签来进行标签推荐。在Stack Overflow数据中，只有问题才有标签，因此当在❷处建立查询参数时，使用了一个过滤查询来限制返回的结果为问题（postTypeId为1）。你也会注意到，该查询请求返回排名最高的10篇文档，可以尝试

不同的结果数目来确定你数据上的最优推荐结果。

当在**③**处建立将发送给Solr server的请求时，使用一个定制的MoreLikeThisRequest而不是标准的Solr QueryRequest。MoreLikeThisRequest将使用HTTP POST来将可能的长查询文档直接传递给Solr/mlt查询handler。该请求的内容参数用于保存推荐的目标内容。

现在已经获得了结果，必须要从中抽取标签并排序来提供推荐。在**④**处收集每个标签的次数并按照打分对标签排序。ScoreTag类用于存储结果集合中每个发现的标签及其出现的次数和得分。我们可以在清单7-22中更进一步了解这一点。

清单7-22 收集标签并对它们排序

```
protected ScoreTag[] rankTags (SolrDocumentList documents,
                                int maxTags) {
    OpenObjectIntHashMap<String> counts =
        new OpenObjectIntHashMap<String> ();

    int size = documents.size ();
    for (int i=0; i < size; i++) {
        Collection<Object> tags = documents.get (i) .getFieldValues ("tags");
        for (Object o: tags) {
            counts.adjustOrPutValue (o.toString (), 1, 1);
        }
    }
    maxTags = maxTags > counts.size () ? counts.size () : maxTags;
    final ScoreTagQueue pq = new ScoreTagQueue (maxTags);
    counts.forEachPair (new ObjectIntProcedure<String> () {
        @Override
        public boolean apply (String first, int second) {
            pq.insertWithOverflow (new ScoreTag (first, second));
            return true;
        }
    });
    ScoreTag[] rankedTags = new ScoreTag[maxTags];
    int index = maxTags;
    ScoreTag s;
    int m = 0;
    while (pq.size () > 0) {
        s = pq.pop ();
        rankedTags[--index] = s;
        m += s.count;
    }
}
```

← ① 对标签计数

← ② 对标签排序

← ③ 收集排序的标签

```
}  
for (ScoreTag t: rankedTags) {  
    t.setScore (t.getCount () / (double) m);  
}  
return rankedTags;  
}
```

← ④ 对标签打分

上述排序和评分过程一开始要对标签计数。在①处对结果集中的文档进行扫描，从tags字段中抽取标签并对每个标签计数。

当所有标签都有计数值时，就可以按照这些值对标签排序。在②处，会在一个保存最频繁标签集的优先级队列中收集标签集合。在③处从优先级队列中抽取结果并在④计算标签的得分。每个标签的得分是结果集截断（即去掉低频标签）之后该标签出现结果的次数除以集合中标签出现的总数。这样，每个标签的得分都在0到1之间，该值越接近1，表明该标签在结果集中越重要。一个标签集合中，如果排名靠前的标签得分更高，那么也意味着会返回一个较小的标签集合，而如果得分较低也意味着会从结果集中返回多个标签。这也可以用作确信度的一种度量方法。通过实验可以得到一个截断得分值，低于该值的标签不宜被推荐。

7.6.5 对标签推荐系统进行评估

评估标签推荐系统与前面评估分类器的输出本质上并没有太大区别。你会使用一部分没有用于训练的Stack Overflow数据作为测试集，并对该集合中的每个问题推荐标签，并将这些推荐标签与已分配标签进行比较。这里与分类的一个显著的不同在于：每篇训练文档都有多个标签，每个查询也会推荐多个标签。因此，这里的测试会得到两种得分：第一种得分是至少有一个标签正确的测试文档数目。至少有一个标签正确可以让你确定分类器怎样才算正确。第二种得分是至少50%以上标签正确的文档数目。例如，如果一篇测试文档分配了4个标签，而推荐产生的标签集合中包含了这4个标签中的至少2个，那么这篇文档才被判定为正确。上述做法能够让你对匹配条件更严格时精度降低有所理解。

除了上述指标之外，可以为在训练和测试数据中遇到的标签子集计算正确比率。可以识别测试集中频率最高的那些标签，并为这些标签生成独立的精度指标。

下面通过代码来深入了解上述计算的过程。首先必须要关注从XML文件中抽取

Stack Overflow数据。为实现这一功能使用StackOverflowStream类。该类使用StAX API来解析XML文档并产生包含每个帖子字段的StackOverflowPost对象，这些字段包括title、body和tags等。StackOverflowStream的很多代码都与XML解析以及在多个贴子上迭代的代码类似，因此我们这里并不重复这一代码，它们的完整代码可以在本书附带的样本代码中找到。

为了给单独的标签构建指标，必须要抽取一个标签集合来从Stack Overflow数据中获得数据。下列命令将从测试集中抽取最频繁的25个标签，其中出现次数少于10个帖子的标签会被去掉。

```
$TT_HOME/bin/tt countStackOverflow \  
  --inputFile stackoverflow-test-posts.xml \  
  --outputFile stackoverflow-test-posts-counts.txt \  
  --limit 25 --cutoff 10
```

最后的结果将是包含三列的文本文件，这三列分别是排名、次数和标签。下面的结果摘录给出了输出的一部分，其中c#是出现次数最多的标签，它在1480个帖子中出现，其次是.net，出现次数为858，而asp.net和java的出现次数分别是715和676：

1	1480	c#
2	858	.net
3	715	asp.net
4	676	java

现在已经知道了类别的计数值，于是可以将它们输入到测试过程中。下列命令用于执行测试类TestStackOverflow。该类会读入文本数据并从Stack Overflow XML格式中抽取必要的字段，然后使用TagRecommenderClient从Solr服务器请求一系列标签，最后将分配给测试数据上的标签与推荐的标签进行对比。在运行时，该类会产生描述推荐系统如何运行的指标。

```
$TT_HOME/bin/tt testStackOverflow \  
  --inputFile stackoverflow-test-posts.xml \  
  --countFile stackoverflow-test-posts-counts.txt \  
  --outputFile stackoverflow-test-output.txt \  
  --solrUrl http://localhost:8983/solr
```

当testStackOverflow运行时，每处理100个帖子它就会生成一次指标。这样就可以

在运行评估程序时让你知道推荐系统在测试帖上的效果。下面的摘录给出了在标注300篇和400篇测试文档时的推荐效果。

```
evaluated 300 posts; 234 with one correct tag, 151 with half correct
  %single correct: 78, %half correct: 50.33
evaluated 400 posts; 311 with one correct tag, 204 with half correct
  %single correct: 77.75, %half correct: 51
```

这里，所有标注文档中的77%到78%有单个标签标注正确，大约50%的文档有一半或更多的标签标注正确。你会注意到，随着标注的文档增多，这些百分比指标也会趋于稳定。这也意味着尽管Stack Overflow的测试数据对10 000篇文档进行测试可能有些过分。这种情况下，利用更少的文档进行测试可能更合理。

当testStackOverflow结束时，它也会将单独标签的指标导出到指定的输出文件中。该文件包含最终的单标签百分比和半标签百分比正确率以及在counts文件中发现标签的正确性指标：

```
evaluated 10000 posts; 8033 with one correct tag, 5316 with half correct
  %single correct: 80.33, %half correct: 53.16
-- tag    total    correct pct-correct --
networking    48      12      25
nhibernate    70      48      68
visual-studio 152     84      55
deployment    48      19      39
```

对每个单独的标签，可以对出现该标签的测试文档数目计算，然后计算该标签推荐给某篇特定文档的次数。根据上述值可以推导出推荐系统在特定标签上的正确率。上述情况下，推荐系统在标签networking上的表现不太好，但在nhibernate上的表现却相当不错。如果变换推荐系统的训练数据，可以跟踪这些针对单个标签的指标的变化情况。

没有任何理由非要坚持前X个标签。如果对推荐系统在其他标签上的执行效果感兴趣的话，那么可以通过手工修改counts文件来纳入这些标签。只要标签在测试集中存在，就可以在输出文件中看到针对它们的指标的输出结果。

7.7 小结

本章考察了分类算法自动对文本文档进行分类和标注的一些方法。我们依次讨

论了自动分类器构建的过程：准备输入、为产生文档分类模型来训练分类器、测试分类器来评估分类器的结果质量以及分类器部署于生产环境。我们给出了训练分类器时选择合适分类机制以及正确特征的重要性，并探讨了获取测试数据的技术，包括使用公开可用资源的方式，利用已有类别集合来对训练集进行自举的方式以及通过人工判定来创建训练集的方式。我们探讨了一些不同的评估指标，讨论了从不同角度来给出分类算法性能的精确率、正确率、召回率和混淆矩阵等指标。我们探讨了对输入或控制算法的参数进行修改来提高分类器结果的方法，并展示了为了在生产环境中提供分类功能而将每种分类器集成到Solr中的方式。

本章也介绍了分类和文本分类中的基本概念，这样你就可以开始对其他算法的探索历程。当你对现有研究和可用软件进行调研之后就会发现，有大量方法可以解决文本分类问题。现在我们只讨论了其中的一些基本算法，包括kNN、朴素贝叶斯和最大熵，现在的知识基础可以允许你探索利用分类和标注来驾驭文本的方法。不论是对开发者还是研究者而言，都存在大量其他的选择。每种算法都有自己不同的特性，这些特性使得它们适合于应用的需求。下面给出了除本章例子之处需要考虑的问题。

Mahout项目也包括一个logistic回归算法，其使用了随机梯度下降作为学习技术。一般而言，这与OpenNLP中现存的logistic回归分类器类似，但是Mahout中的实现上使用了多种特征解释的方式，这一点十分有趣，它提供了一种机制能在单个模型中同时融入数字型、词语型和文本型特征。Ted Dunning领导了Mahout中该分类器的实现，具体描述在Manning出版社出版的另一本书 *Mahout in Action* (2011) 中有详细介绍。

支持向量机 (Support Vector Machines, 简称SVMs) 已广泛用于文本分类。大量研究覆盖了利用SVM对文本建模的方方面面，许多开源实现也用于实现基于SVM的文本分类系统。这些开源实现包括Thorsten Joachims的SVM-LIGHT (<http://svmlight.joachims.org>) 和Chih-Chung Chang与Chih-Jen Lin的LIBSVM (<http://www.csie.ntu.edu.tw/~cjlin/libsvm>)。通过上述每一个库，文本分类可以使用多种语言来实现。

可以进一步探讨的还有不计其数的其他算法变形以及这些变形算法和其他技术的组合。本书第9章《未驾驭的文本》中会介绍其中的一些方法。

7.8 参考文献

Lewis, David; Yang, Yiming; Rose, Tony; Li, Fan. 2004. "RCV1: A New Benchmark Collection for Text Categorization Research." *Journal of Machine Learning Research*, 5:361-397. <http://www.jmlr.org/papers/volume5/lewis04a/lewis04a.pdf>.

Owen, Sean; Anil, Robin; Dunning, Ted; Friedman, Ellen. 2010. *Mahout in Action*. Manning Publications.

Rennie, Jason; Shih, Lawrence; Teevan, Jaime; Karger, David. 2003. "Tackling the Poor Assumptions of Naive Bayes Text Classifiers." <http://www.stanford.edu/class/cs276/handouts/rennie.icml03.pdf>. "Americans Spend Half of Their Spare Time Online." 2007. Media-Screen LLC. <http://www.media-screen.com/press050707.html>.

第8章 构建示例问答系统

本章内容

- 应用文档自动标注技术
- 在搜索中使用文档和子文档的标签
- 基于附加准则对Solr返回的文档进行重排序
- 针对用户问题生成可能的答案

在前面各章中，我们独立地考察了多种不同的技术和方法。尽管我们已经设法构建了只集中关注一两项技术的有效应用，但通常情况下还是需要组合目前介绍的多个工具来完成任务。例如，当在帮助用户寻找和发现与他们的信息需求相关的新内容时，多面搜索和标注（分类）会自然融合，内容聚类 and 搜索也是如此。就本章的目的而言，你将构建一个问答（QA）系统，它能够利用搜索、命名实体识别、字符串匹配及其他技术来回答用户提出的基于事实的问题（用英语描述）。尽管其他各章都自成体系，即不需要读者了解其他章的内容，但是本章却假设读者已经阅读并了解前面的各章，因此我们在这里不再解释Solr和其他系统的基本知识。

在构建问答系统之前，首先回顾一下前面的内容。回顾之余你会明白这些内容构成了本章的概念基础。第1章中，我们讨论文本对不同应用的重要性，介绍了一些搜索和自然语言处理领域的基本术语，并给出了构建这类系统所面临的挑战。即使不用特别指明，大部分这些基础知识都将同时显式或隐式地在本章使用。

第2章集中关注文本处理的基础知识，包括词性标注、句法分析、文法知识等，

这些可能会让你回到高中时代。我们还花时间了解了如何从原始格式获得内容并转换为Apache Tika所需要的格式。尽管我们在本章示例中没有显式使用Tika，但我们将对内容进行预处理并转换成本章任务所需的格式。我们也将大量使用内容切词、句法分析和词性标注工具来回答问题。

第3章主要介绍搜索，并介绍了Apache Solr这个强大的搜索平台，利用这个平台可以快速方便地索引文本并通过查询来返回文档。本章我们再次利用Solr作为问答系统的支撑平台，并利用Apache Lucene的多个高级功能。

第4章主要介绍模糊字符串匹配，该技术对于很多日常的文本处理工作十分有用。本章利用那一章学到的技术来进行自动拼写校正，也会利用诸如 n 元组等一些其他的模糊字符串技术。这其中一些字符串技术会用于Lucene的低层，可以很容易将一个拼写校正模块嵌入到我们的系统中，尽管我们并没有选择这样做。

第5章使用OpenNLP来识别并对文本中的专有名词进行分类。本章将再次使用OpenNLP来完成该项任务，同时识别短语。这种做法不论是对分析查询还是处理用于寻找答案的内容来说都有用。

第6章介绍了聚类，并展示如何能够利用无监督技术将相似文档自动分组。尽管本章当中没有展示这一点，但是在寻找答案和确定结果中的近似重复时，聚类技术可以用于缩小搜索空间。

最后，第7章展示如何对文本分类并利用分类器来自动将关键词或者社会化标签赋予新文本。本章也将利用该技术来将新出现的问题归到某个类别中。

现在对所学的东西已经有所感觉了，下面就将这些东西融合起来构建一个实际的应用。我们将构建一个示例QA系统，其目的是为了展示多少迄今为止讨论过的技术可以组合在一起构成一个实际工作系统。我们将利用维基百科作为知识库来构建一个回答事实性问题的简单QA应用。为达到此目标，我们将利用Solr作为基准系统，这一方面是因为它能够进行段落检索，另一方面是因为它的插件式架构便于扩展。从该基准系统出发，可以在索引构建过程中插入分析功能，同时在搜索端功能中插入对于用户自然语言问题的分析能力，从而对答案进行排序并返回结果。下面开始讨论问答系统及一些相关应用。

8.1 问答系统基础知识

如同名字所暗示的那样，问答（question answering, QA）系统被设计成在提出自然语言问题（如 Who is the President of the United States?）的情况下给出答案。QA系统能够减轻最终用户通过页面和搜索结果页面搜索或者按多面方式点击和浏览页面的需求。例如，IBM Watson DeepQA系统（<http://www.ibm.com/innovation/us/watson/>）使用了一个复杂的QA系统在Jeopardy!（<http://www.jeopardy.com>）节目上和人类竞赛。大家有没有注意到它赢了有史以来最强的两个Jeopardy!玩家？该系统使用了大量的计算机基于大规模知识库及辅助的竞赛策略系统（包括选择线索、投注等功能，参考图8-1）来处理答案（记住，Jeopardy!要求“答案”表示成问题的格式）。

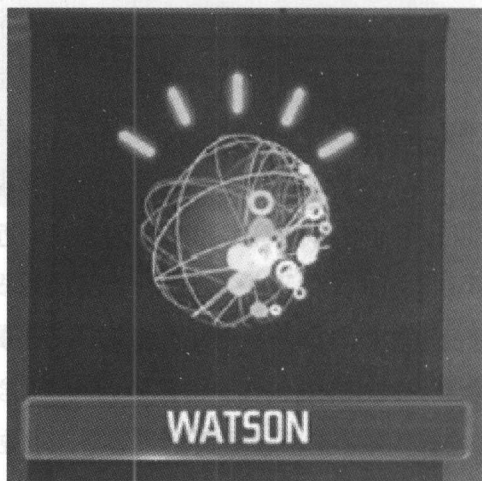


图8-1 出现在Jeopardy! IBM挑战赛上的IBM Watson Avatar系统的截屏图

需要注意的是，不要把自动的QA系统和当前Web上流行的众包型QA系统如Yahoo! Answers或ChaCha混为一谈，尽管后者中的一些技术对构建自动问答系统也有用。问答系统在很多方面与搜索很类似，二者都是提交查询；这些查询通常包含一些关键词，然后通过浏览返回的文档或页面来寻找答案。在问答系统中，通常输入的是一个完整的句子而不只是关键词。为了使答案更加具体，用户期望返回的是比文档小很多的一小段文本。一般而言，问答系统很难，但是对于特定应用或类型来说，问答系统可以十分有效。很多问题的答案十分复杂，需要进行大量理解之后才

能作答。正因如此，这里我们构建的问答系统并不具备完全理解的能力，而是构建一个面对事实型问题（如Who is the President of the United States?）时比标准搜索系统更好的一个系统。

IBM的Waston：不只是Jeopardy!

IBM的Waston系统在Jeopardy! 节目中展示是一种吸引大家关注的方式，但是其更深的意图显然不止是在Jeopardy!上竞赛，而是帮助人们更加快速高效地筛选信息。以下内容引用自IBM网站¹：

DeepQA技术给人们提供了一种强大的信息收集和决策支持工具。对于最终用户来说一个典型的场景就是用自然语言形式输入问题，这和平时问其他人问题没有什么两样。系统会通过大量可能的证据上进行筛选来返回最有说服力、最精确的答案排序列表。这些答案包括证明或支持的证据，这样用户就可以对这些证据进行快速评估从而选择正确答案。

上述系统的深度超过我们这里讨论的范围，感兴趣的读者可以去了解IBM的DeepQA项目（参考<http://www.research.ibm.com/deepqa/deepqa.shtml>）以获取更多内容。

一个完备的QA系统可能会试图回答不同类型的问题，这些问题覆盖范围从事实型问题到更深奥的问题。也要记住的是，尽管大部分QA系统都尽力返回很短的答案，但对于一个QA系统来说，返回多个段落甚至多篇文档作为答案也是完全合理的。比如，一个极其高端的系统（就作者所知，目前还不存在这样的系统）可能可以回答需要更深分析和响应的问题，比如“What are the pros and cons of the current college football bowl system?”（当前大学橄榄球赛制的优点和缺点如何？）或“What are the short- and long-term effects of binge drinking?”（酗酒的短期和长期影响如何？）

对更深的事实型问答系统进行挖掘可以想象成词和短语层面上的模糊匹配问题。正因为如此，这里的方法和前面记录匹配中所用的方法有些类似，所不同的只是这里需要集中理解给定问题的答案类型。例如，如果用户问“Who is the President

¹ 2011年4月12日的<http://www.research.ibm.com/deepqa/faq.shtml>

of the United States?”，那么可以预期问题的答案是个人名，而当用户问“What year did the Carolina Hurricanes win the Stanley Cup?”，答案就应该是一个年份。在深入到系统构建之前，我们花点时间构建一些代码以便你能照着这些代码入手。

8.2 安装并运行QA代码

前面提到，我们会在Solr基础上构建系统，因此安装并运行QA代码意味着利用已有的Solr封装即可，这很像前面聚类那一章的做法。对于本例，我们会使用一种不同的Solr设置。如果没有的话，可以按照GitHub上的README文件（<https://github.com/tamingtext/book/blob/master/README>）中的说明来操作。接下来从目录TT_HOME下运行./bin/start-solr.shsolr-qa。如果顺利的话，就可以从浏览器访问地址<http://localhost:8983/solr/answer>并得到一个简单的QA界面。利用正在运行的系统，可以加载一些内容以便能够回答一些问题。

正如可以想象到的那样，由于构建在搜索引擎之上的QA系统（大部分QA系统都是这样）并没有对所有问题和答案的本质认识，因此需要搜索引擎中的内容来作为发现答案的信息源。例如，如果给引擎加载写于克里斯多弗·哥伦布（ChristopherColumbus）之前时期的欧洲文档（当然它们要数字化，对吧？）并问系统“What shape is the Earth?”（地球是什么形状？），系统可能会返回与扁平相当的答案。对于我们的系统而言，我们将使用2010年10月11日导出的英语维基百科语料（前100K文档缓存于<http://maven.tamingtext.com/freebase-wex-2011-01-18-articles-first100k.tsv.gz>，该压缩包的大小为411MB）。注意该文件很大，但是为了展示实际数据上的应用这也是必需的。下载该文件之后，利用gunzip或类似的工具解压。如果觉得该文件太大或者首先想尝试一个更小的版本，可以下载<http://maven.tamingtext.com/freebase-wex-2011-01-18-articles-first10k.tsv>，该文件由前面大文件中的前1万篇文章构成。该文件没有压缩，因此不需要解压。

- 获得数据之后，可以通过下列步骤来对其进行索引。
- 键入 `cd $TT_HOME/bin。`
- 在*NIX系统下运行 `indexWikipedia.sh --wikiFile<PATH TO WIKI FILE>`或在Windows系统下运行 `index-Wikipedia.cmd --wikiFile<PATH TO WIKI FILE>`。这需要一段时间来完成。可以使用--help 选项来了解所有可用的索引选项。

构建索引之后，就准备好探索QA系统了。可以通过在浏览器上输入 `http://localhost:8983/solr/answer` 来访问QA系统，我们使用了Solr内置的VelocityResponseWriter搭建了一个简单的QA用户界面，该VelocityResponseWriter接收Solr的输出并对其应用Apache Velocity模板（`http://velocity.apache.org`）。Velocity是一个主要用于构建Java应用支持网站的模板引擎。如果上述步骤都顺利的话，你会看到类似图8-2给出的屏幕截图。假设上述所有过程都正确运行，下面就开始讨论系统构建的架构和代码。

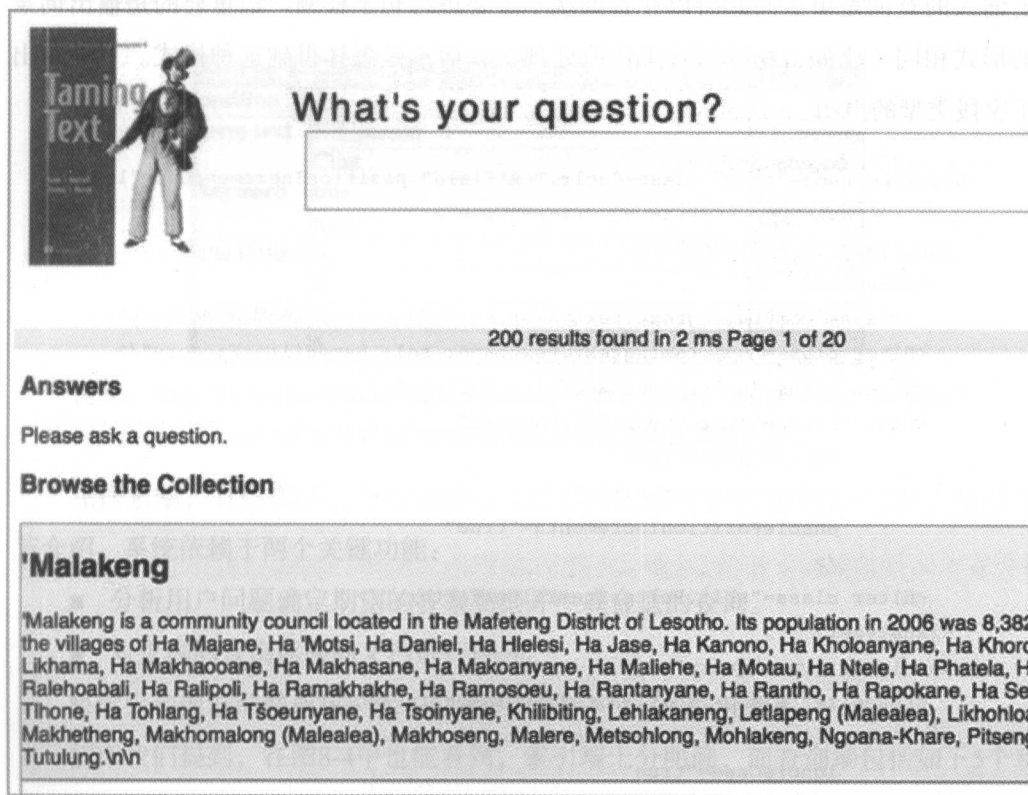


图8-2 本书所带的基于事实的QA系统的截图

8.3 一个示例问答系统的架构

和前面的搜索很像，我们的QA系统必须要进行索引的构建、搜索和结果的后处理。在索引端，大部分定制都集中于分析过程。我们已经创建了两个Solr分析插件，

一个用于句子检测，另一个用于命名实体识别。两个插件都基于OpenNLP来实现想要的功能。与Solr中的大部分分析过程不同，我们选择直接切分成句子，这样就允许将那些句子直接发给命名实体词条过滤器，从而避免在Solr和OpenNLP中各一次的额外切词过程。由于前面已经介绍过句子检测和命名实体识别，这里给出相关的类（SentenceTokenizer.java和NameFilter.java）并给出schema.xml（位于solr-qa/conf目录下，为了节省空间，部分内容做了编辑处理）中文本字段类型的声明。注意这里我们违反了在索引构建和搜索中要进行相同的分析处理的一般规则，这是因为我们假定输入的查询是单个问题所以在查询端不需要进行句子检测。最重要的是输出词条的形式相同（比如，相同的词干还原处理），而不是怎样得到这种格式。下面给出了字段类型的声明。

```
<fieldType name="text" class="solr.TextField" positionIncrementGap="100"
  autoGeneratePhraseQueries="true">
  <analyzer type="index">
    <tokenizer
      class="com.tamingtext.texttamer.solr.SentenceTokenizerFactory"/>
    <filter class="com.tamingtext.texttamer.solr.NameFilterFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="stopwords.txt"
      enablePositionIncrements="true"
    />
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="stopwords.txt"
      enablePositionIncrements="true"
    />
    <filter class="solr.WordDelimiterFilterFactory"
      generateWordParts="1" generateNumberParts="1"
      catenateWords="0" catenateNumbers="0" catenateAll="0"
      splitOnCaseChange="1"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.PorterStemFilterFactory"/>
```



```
</analyzer>
</fieldType>
```

尽管由于前面已经介绍过分析过程，这里对此有所忽略，但重要的一点是注意到NameFilterFactory会输出原始的词条和表示任意命名实体的词条。这些命名实体词条出现的位置与原始词条的位置一样。例如，通过Solr的analysis.jsp页面（<http://localhost:8983/solr/admin/analysis.jsp>）运行句子“Clint Eastwood plays a cowboy in *The Good, the Bad and the Ugly*.”之后就会产生总共4个词条，这些词条占据了输出结果中前面的2个位置（每个位置上有两个词条），如图8-3所示。

com.tamingtext.texttamer.solr.NameFilterFactory (luceneM		
position	1	2
term text	NE_person	NE_person
	Clint	Eastwood
keyword	true	true
	true	true
startOffset	0	6
	0	6
endOffset	5	14
	5	14

图8-3 句子“Clint Eastwood plays a cowboy in *The Good, the Bad and the Ugly*.”中命名实体词条如何与原始词条的位置发生重叠的例子

在搜索端，有很多活动部件需要编写更多的代码，这些代码的细节将在后续小节介绍。系统依赖于两个关键功能：

- 分析用户问题确定期望的答案类型并生成合适的查询。
- 对生成的查询返回的文档进行评分。

将索引和搜索部件连在一起，就得到如图8-4所示的系统架构。

前面我们提到，在图8-4中也能看到，索引端十分明确，而查询端包括如下5个步骤。

- 分析用户查询。
- 确定答案的类型以便在搜索时可以找到候选最佳答案。
- 利用分析后的查询和答案类型生成Solr/Lucene查询。
- 执行搜索过程来确定候选的可能包含答案的段落。
- 对答案排序并返回结果。

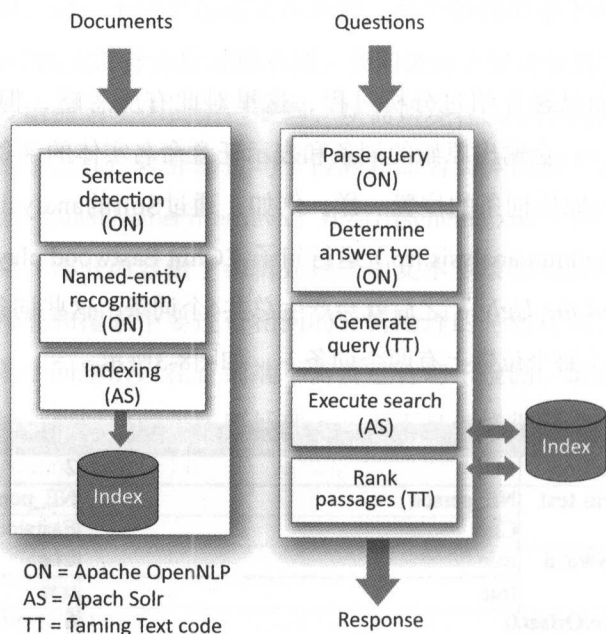


图8-4 一个构建于Apache Solr、OpenNLP及部分自己编写的段落排序代码之上的示例问答系统的架构

结合上下文理解这5个步骤，我们将这些步骤分成两部分：理解用户的查询和对候选段落进行排序。我们利用Solr中定义明确的插件机制来实现这两个部分。

- 一个QParser（和QParserPlugin）：处理输入的用户查询并创建Solr/Lucene查询；
- 一个SearchComponent：与其他SearchComponent串在一起构建合适的响应。

通过第3章可以了解更多背景。

接下来马上近距离考察代码，但是现在值得花点时间回顾一些搜索和Solr的有关章节（第3章），并且阅读一些Solr文档来了解这两个部分在Solr中的工作机制。假设你对相关机制比较了解，下面看看如何理解用户的问题。

8.4 理解问题并产生答案

我们关注利用用户问题并产生答案的三个部件，它们主要围绕答案类型（AT）确定、利用AT来生成有意义的搜索引擎查询以及最终对查询返回结果进行排序。

其中，确定答案类型是问题的关键所在，而之后的查询生成和段落排序相对比较简单。在我们给出的示例中，答案类型涉及三个部分：训练、组块和真正的答案类型确定。上面这三个部分以及给定答案类型下的查询生成和段落排序方法，会在后续小节中进行概述。在我们的示例中，需要指出的一点是，我们假定用户输入的是自然语言表述的问题，比如“Who was the Super Bowl MVP?”，而不是像第3章给出的布尔逻辑查询。由于我们会训练一个分类系统来确定给定问题的答案类型，而分类所用的训练数据则是基于自然语言问题来进行的，所以上述这一点十分重要。

8.4.1 训练答案类型分类器

对于本章给出的示例系统，训练数据（位于源码的dist/data目录下）由1888个问题组成，每个问题由Tom Morton手工进行了标注，这是其博士论文工作的一部分（参考Morton [2005]）。训练问题看起来像下面给出的例子：

- P Which French monarch reinstated the divine right of the monarchy to France and was known as “The Sun King” because of the splendour of his reign?
- X Which competition was won by Eimear Quinn with “The Voice in 1996, ” this being the fourth win in five years for her country?

在训练问题中，第一个字符表示的是问题答案的类型，而剩余部分是问题本身。我们的训练数据支持很多不同的答案类型，但为了简单起见我们的示例系统只处理其中的4种类型（位置、时间、人物和组织）。表8-1给出了训练数据中所支持的答案类型及相应的样例。

表 8-1 训练数据答案类型

答案类型（训练代码）	样 例
人物（P）	Which Ivy League basketball player scored the most points in a single game during the 1990s?
位置（L）	Which city generates the highest levels of sulphur dioxide in the world?
组织（O）	Which ski resort was named the best in North America by readers of <i>Conde Nast Traveler</i> magazine?
时刻（T）	What year did the Pilgrims have their first Thanksgiving feast?
时长（R）	How long did <i>Gunsmoke</i> run on network TV?

(续表)

答案类型 (训练代码)	样 例
货币 (M)	How much are Salvadoran workers paid for each \$198 Liz Claiborne jacket they sew?
比率 (C)	What percentage of newspapers in the U.S. say they are making a profit from their online site?
数量 (A)	What is the lowest temperature ever recorded in November in New Brunswick?
距离 (D)	What is the approximate maximum distance at which a clap of thunder can be heard?
描述 (F)	What is dry ice?
标题 (W)	In which fourteenth-century alliterative poem by William Langford do a series of allegorical visions appear to the narrator in his dreams?
定义 (B)	What does the postage stamp cancellation O.H.M.S. mean?
其他 (X)	How did the banana split originate?

为训练答案类型分类器，我们利用AnswerTypeClassifier类的主方法，如下。

```
java -cp -Dmodels.dir=<Path to OpenNLP Models Dir>
        -Dwordnet.dir=<Path to WordNet 3.0> \
        <CLASSPATH>com.tamingtext.qa.AnswerTypeClassifier \
        <Path to questions-train.txt><Output path>
```

下面我们将跳过测试这一步，虽然测试通常总是与分类器模型构建相关联，但是Tom在他的论文中已经进行了测试。如果你对测试过程感兴趣的话，可以参考第5章和第7章有关分类的相关内容。

训练过程的代码相当简单，首先基于OpenNLP对训练集进行组块分析（浅层分析）和词性标注，然后将结果输送给OpenNLP的MaxEnt分类器。关键的代码片段如清单8-1所示。答案类型模型的训练过程与前面章节在OpenNLP上的训练过程（命名实体识别和标注）十分类似。

清单8-1 训练问题模型

```
AnswerTypeEventStream es = new AnswerTypeEventStream (trainFile,
        actg, parser);
GISModel model = GIS.trainModel (100,
```

```
newTwoPassDataIndexer (es, 3) );
newDoccatModel ("en", model) .serialize (
    newFileOutputStream (outFile));
```

使用事件流来输入
训练样例，利用
OpenNLP 的 MaxEnt
分类器进行训练

模型训练完之后，必须要编写一些代码来使用该模型。为实现这一点，我们编写了一个名为QuestionQParser的Solr Qparser（及工厂类QParserPlugin）。在深入到代码之前，QuestionQParser的配置如下：

```
<queryParser name="qa" class="com.tamingtext.qa.QuestionQParserPlugin"/>
```

正如你看到的那样，Solr主要关注的配置不是QParser本身而是QParserPlugin。

QuestionQParserPlugin主要做的事情是加载AT模型并构建一个QuestionQParser。同对任意QParser及其关联工厂一样，我们想在QParserPlugin而不是QParser自身的初始化中执行任意大花销或一次性的计算，这是因为QParser在每次请求中都会构建一次，而QParserPlugin只会构建一次（每次提交时才构建）。对于QuestionQParser，初始化代码负责加载AT模型和另一个资源WordNet。整个代码就是一段很清楚的初始化代码，如清单8-2所示。

清单8-2 初始化代码

```
public void init (NamedListinitArgs) {
    SolrParamsparams = SolrParams.toSolrParams (initArgs) ;
    String modelDirectory = params.get ("modelDirectory",
        System.getProperty ("model.dir")) ;
    String wordnetDirectory = params.get ("wordnetDirectory",
        System.getProperty ("wordnet.dir")) ;
    if (modelDirectory != null) {
        File modelsDir = new File (modelDirectory) ;
        try {
            InputStreamchunkerStream = new FileInputStream (
                new File (modelsDir, "en-chunker.bin")) ;
            ChunkerModelchunkerModel = new ChunkerModel (chunkerStream) ;
            chunker = new ChunkerME (chunkerModel) ;
            InputstreamposStream = new FileInputStream (
                new File (modelsDir, "en-pos-maxent.bin")) ;
            POSModelposModel = new POSModel (posStream) ;
            tagger = new POSTaggerME (posModel) ;
            model = new DoccatModel (new FileInputStream (
                new File (modelDirectory, "en-answer.bin")))
                .getChunkerModel () ;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

模型目录下
包含了本书
所用的所有
OpenNLP 模型

WordNet 是
用于辅助确
定答案类型
的词汇资源

数据库块工具
和 Parser 一
起对问题进行
浅层分析

构建真正模型
并保存该模型
以备重用，这
是因为它是线
程安全的，但
是包含的类并
不一定安全

Tagger 负责
词性标注

```

        probs = new double[model.getNumOutcomes () ];
        atcg = new AnswerTypeContextGenerator (
            new File (wordnetDirectory, "dict"));
    } catch (IOException e) {
        throw new RuntimeException (e) ;
    }
}
}

```

WordNet (<http://wordnet.princeton.edu/>) 是普林斯顿大学面向英语和其他语言构建的一项词汇资源，它包含了有关词的信息，如同义词、反义词、上位词、下位词以及其他有关单个词的有用信息。它允许用于商业用途。你会看到它有助于我们更好地理解问题。

给定这些资源的构建之后，工厂的主要任务就是构建我们的QuestionQParser，构建代码如清单8-3所示。

清单8-3 构建QuestionQParser

```

@Override
public QParser createParser (String qStr, SolrParams localParams,
                             SolrParams params,
                             SolrQueryRequest req) {
    answerTypeMap = new HashMap<String, String> ();
    answerTypeMap.put ("L", "NE_LOCATION");
    answerTypeMap.put ("T", "NE_TIME|NE_DATE");
    answerTypeMap.put ("P", "NE_PERSON");
    answerTypeMap.put ("O", "NE_ORGANIZATION");
    QParser qParser;

```

构建我们感兴趣的
答案类型的
map，这些类型
比如位置、人物、
时间和日期

当用户输入的不是问题或
者输入 `*:*` 查询（意味着
匹配所有文档）时，利用
这条 if 子句来构建一个常
规的 Solr 查询分析器

构建负责分析
用户问题的组
块分析器

```

    if (params.getBool (QParams.COMPONENT_NAME, false) == true
        && qStr.equals ("*:*") == false) {
        AnswerTypeClassifier atc =
            new AnswerTypeClassifier (model, probs, atcg) ;
        Parser parser = new ChunkParser (chunker, tagger) ;
        qParser = new QuestionQParser (qStr, localParams,
            params, req, parser, atc, answerTypeMap) ;
    } else {
        //just do a regular query if qa is turned off
        qParser = req.getCore ().getQueryPlugin ("edismax")
            .createParser (qStr, localParams, params, req) ;
    }
}

```

输入用户问题和
在 init 方法中预初
始化的资源以构建
QuestionQParser

Answer Type Classifier
分类器使用训练好的答
案类型模型（位于模型
目录下）来对问题进行
分类

```
return qParser;  
}
```

上述代码中主要的关注区域是答案类型map和QuestionQParser的构建。答案类型map包括从AnswerTypeClassifier产生的内部代码（参考表8-1）到索引实体类型的映射。例如，L映射为NE_LOCATION，这和我们在用NameFilter类进行索引的过程中对位置型命名实体标注的方式一样。后面会利用这个map在Solr查询中构建合适的子句，而QuestionQParser是实际分析问题和建立Solr/Lucene查询的类。关于这一点，我们剥开QuestionQParser的一层外衣对其进行更加深入的考察。

QuestionQParser负责三件事，这些事情都在类的parse方法中处理：

- 对查询进行组块分析生成Parse对象。
- 计算答案类型。
- 将Solr/Lucene查询构建为SpanNearQuery（马上会介绍有关它的更多内容）。

8.4.2 对查询进行组块分析

组块是一种轻量级的分析（这里的分析有时称为深度分析）形式，当集中关注句子中的关键片段（如动词和名字短语）而忽略其他部分时，它有助于节省CPU周期，而这样做完全和我们要达到的目标相匹配。对于问题我们并不需要深度分析，而只需要得到问题的关键部分即可。QParser中的分析代码可以写成如下的一行：

TreebankParser
分析问题，然后
果 Parse 对象
分类器用于确
答案类型

```
Parse parse = ParserTool.parseLine(qstr, parser, 1)[0];
```

注意到在上述分析样例中，我们传入的是一个Parser引用。该引用是一个ChunkParser的实例，我们编写该实例来实现OpenNLP的Parser接口。ChunkParser通过使用OpenNLP的TreebankChunker为提交的问题构建了一个浅层Parse对象，而正如名字某种程度上暗示的那样，TreebankChunker使用CoNLL2000会议任务中的宾州树库（参考<http://www.cnts.ua.ac.be/conll2000/chunking/>）和一个ParserTagger创建一个Parse对象。ParserTagger负责标注问题中词的词性。该类对于chunker来说是必备的，这是因为chunker模型需要使用词性信息来训练。换句话说，本例中的词性标签是组块分析所必需的特征。直观上看，这是合理的，如果能够首先识别句子中所有的名词，那么识别名词短语就更容易。在上述样例代码中，我

们给词项标注器配以一个叫作tag.bin.gz的在OpenNLP中现存的模型。类似的，TreebankChunker实例使用了下载模型中的EnglishChunk.bin.gz模型来接受词性标注器的输出并产生分析结果。尽管需要大量工作（这些工作一起都在单个模型中实现），但是它能够为我们提供估计用户所寻找的答案类型的能力，下面我们将介绍这一点。

8.4.3 计算答案类型

下一步是识别答案类型，然后查找从内部缩略答案类型代码和对命名实体进行索引时所用的标签开始的映射。清单8-4给出了这一部分代码。

清单8-4 识别答案类型

```
String type = atc.computeAnswerType (parse) ;  
String mt = atm.get (type) ;
```

很显然，在AnswerTypeClassifier类及其委托类中做了大量工作，因此在我们讨论Solr/Lucene查询生成之前先了解一下该类。

如名字所暗示的那样，AnswerTypeClassifier是一个接受问题输出答案类型的分类器。从很多方面来说，这是我们的QA系统的关键所在。这是因为如果没有正确的AT，那么寻找既提到所需关键词又包含期望答案类型的段落将十分困难。例如，如果用户问“Who won the 2006 Stanley Cup?”，正确的AT应该显示答案可能是人或组织。那么，如果系统碰到一个包含won、2006及Stanley Cup的段落，那么就可以对该段落排序以确定段落中是否包含正确答案类型的词或短语。比如，系统可能遇到句子“The 2006 Stanley Cup finals went to 7 games.”，这个句子中没有提到任何人或组织，那么系统就会因为该句中不含答案类型而丢弃这个句子。

在构建中，AnswerTypeClassifier加载前面训练出的答案类型模型并且也构建一个AnswerTypeContextGenerator实例。AnswerTypeContextGenerator依赖于WordNet和某些启发式规则确定特征来返回到AnswerTypeClassifier用以分类。AnswerTypeClassifier代码在computeAnswerType和computeAnswerTypeProbs方法中调用了AnswerTypeContextGenerator，整个代码如下所示：

```
public String computeAnswerType (Parse question) {  
    double[] probs = computeAnswerTypeProbs (question) ;
```

通过调用
computeAnswerTypeProbs,
获得某个答案类型的概率

定生成的概率
要求模型给
最佳结果。这
的计算十分简
只是从数组
找到最大概率

```
returnmodel.getBestOutcome (probs) ;
```

```
public double[] computeAnswerTypeProbs (Parse question) {  
    String[] context = atcg.getContext (question) ;  
    returnmodel.eval (context, probs) ;  
}
```

要求 AnswerTypeContextGenerator 返回可以用于预测答案类型的特征列表 (context)

对产生的特征进行评估, 以确定可能的答案类型的概率

本代码的关键是computeAnswerTypeProbs方法中的两行。第一行要求AnswerTypeContextGenerator类从问题分析结果中来选择特征集, 然后第二行将这些特征传递给模型用于评估。模型为每个可能的结果返回一个概率, 然后从中选择概率最大的结果作为答案类型。

正如你在前面章节注意到的那样, 特征选择通常也是个很难的问题, 因此值得详细考察AnswerTypeContextGenerator类的工作流程。AnswerTypeContextGenerator中的特征选择通过getContext () 方法来处理。该方法实现了一些简单规则以基于所问题类型来选择好的特征。这些规则中大部分的前提假设是寻找问题中的关键动词和名词短语, 表述如下:

- 如果疑问词存在 (who、what、when、where、why、how、whom、which、name等)
 - 那么计入疑问词并将之标为qw (qw=who)。
 - 计算疑问词左部的动词, 将其标为动词, 并将其和疑问词一起标记为qw_verb (比如若动词=entered, 那么qw_verb=who_entered)。
 - 将动词右部的所有词计入并将它们标记为rw (如rw=monarchy)。
- 如果焦点名词存在的话 (问题中的关键名词)
 - 将名词短语中的中心词加入并标记为hw (hw=author), 将其词项标记为ht (ht=NN)。
 - 计入修饰该名词的任意词, 并标记为mw (mw=European), 将它们的词项标记为mt (mt=JJ)。
 - 计入单词的任意WordNet同义词集合 (同义词集合指的是某个词的所有同义词), 标记为s (s=7347, 同义词集的ID)。
 - 标识焦点名词是否短语的最后一个名词并标记为fnIsLast (fnIsLast=true)。
- 计入默认的称为def的特征。该默认的特征是一个用于归一化目的的空标识

符。每个问题不管是否选择其他特征都会包含这个标识符，因此它可以为系统提供一个基准特征以供学习。

在讨论上述列表中的关键部分之前，先看看问题“Which European patron saint was once a ruler of Bohemia and has given his name to a Square in Prague?”所选择的特征，如下所示。

```
def, rw=once, rw=a, rw=ruler, rw=of, rw=Bohemia, rw=and,
rw=has, rw=given, rw=his, rw=name, rw=to, rw=a, rw=Square, rw=in,
rw=Prague?, qw=which, mw=Which, mt=WDT, mw=European, mt=JJ, mw=patron,
mt=NN, hw=saint, ht=NN, s=1740, s=23271, s=5809192, s=9505418,
s=5941423, s=9504135, s=23100, s=2137
```

通过运行AnswerTypeTest中的demonstrateATCG测试，可以看到运行的结果。这部分的代码如下。

```
AnswerTypeContextGeneratoratcg =
    newAnswerTypeContextGenerator (
        new File (getWordNetDictionary () .getAbsolutePath ()) ;
    InputStream is = Thread.currentThread () .getContextClassLoader ()
        .getResourceAsStream ("atcg-questions.txt") ;
    assertNotNull ("input stream", is) ;
    BufferedReader reader =
        newBufferedReader (new InputStreamReader (is)) ;
    String line = null;
    while ((line = reader.readLine ()) != null) {
        System.out.println ("Question: " + line) ;
        Parse[] results = ParserTool.parseLine (line, parser, 1) ;
        String[] context = atcg.getContext (results[0]) ;
        List<String> features = Arrays.asList (context) ;
        System.out.println ("Features: " + features) ;
    }
```

再回到特征选择问题，大部分特征可以通过几条简单规则或正则表达式来选择，这一点可以从AnswerTypeContextGenerator类的代码中看到。寻找中心名词的问题要采用其他的规则，这是因为它有一些增加的特征，也是因为我们要做相当多的工作来识别它。中心名词取决于疑问词的类型（who、what、which等），并对于定义要寻找的对象十分重要。例如，在我们给出的有关“ruler of Bohemia”的问题当中，中心名词是 saint，这意味着我们要寻找一个圣人（saint）。于是我们可以使用这个名

字, 利用WordNet来获得可能的同义词, 这些同义词可能有助于识别其他以不同方式来询问相同问题的词。在遍历问题的同时, 我们也应用某些规则来消除有关中心名词的错误匹配。这些规则也基于简单规则和正则表达式。在上述代码中, 该工作大部分都在AnswerTypeContextGenerator的findFocusNounPhrase方法中完成, 由于长度限制, 该方法的代码没有在这里给出。

最后, 记住这里的特征选择过程基于Tom (作者之一) 对问题的分析, 当构建模型时他需要考虑这些重要事项。但并不意味着这是唯一的方法。此外, 给定更多的训练数据, 可能可以让系统来学习模型而不需要任何特征选择过程。从某种程度上说, 这种人在其中的特征选择过程, 是在收集和标注所花费时间及现有查询预先模式分析所花时间之间进行权衡的做法。到底哪一种方式更好取决于拥有的数据和时间。

8.4.4 生成查询

在确定答案类型之后, 需要构建一个查询来从索引中返回候选段落。返回的候选段落必须要满足以下条件才能为QA所用:

- 一个或多个具有正确答案类型的词必须在段落窗口内出现。
- 原始查询的一个或多个关键词项必须在段落窗口内出现。

为了构建能够返回满足上述需求的查询, 必须知道给定文档中的确切匹配位置。在Solr (和Lucene) 中, 完成上述功能的方式是通过SpanQuery及其派生类。具体地说, SpanQuery系列类像Lucene中的其他查询一样来匹配文档, 但是它们也要花费额外的计算时间来访问位置信息, 这样就能通过在位置上迭代来产生更集中的段落而不是更大的文档用于排序。最后, 我们必须特别构建一个SpanNearQuery类来寻找段落, 这是因为我们希望同时找到指定的词项和答案类型。SpanNearQuery类可以构建复杂的基于短语的查询, 这些查询由其他多个SpanQuery实例构成。构建查询的代码如清单8-6所示。

清单8-6 查询生成

```
List<SpanQuery>sql = new ArrayList<SpanQuery> ();  
if (mt != null) {  
    String[] parts = mt.split ("\\|");
```

```

if (parts.length == 1) {
    sql.add (new SpanTermQuery (new Term (field, mt.toLowerCase ()))) ;
} else {
    for (int pi = 0; pi < parts.length; pi++) {
        sql.add (new SpanTermQuery (new Term (field, parts[pi])));
    }
}
}
try {
    Analyzer analyzer = sp.getType ().getQueryAnalyzer ();
    TokenStream ts = analyzer.tokenStream (field,
        new StringReader (qstr));
    while (ts.incrementToken ()) {
        String term = ((CharTermAttribute)
            ts.getAttribute (CharTermAttribute.class)).toString ();
        sql.add (new SpanTermQuery (new Term (field, term)));
    }
} catch (IOException e) {
    throw new ParseException (e.getLocalizedMessage ());
}
return new SpanNearQuery (sql.toArray (new SpanQuery[sql.size ()]),
    params.getInt (QParams.SLOP, 10), true);

```

上述查询生成的代码进行了如下三个步骤。

- 利用一个或多个SpanTermQuery实例将答案类型加到查询中；
- 对给定字段使用查询分析器来为每个词项构建SpanTermQuery实例；
- 构建一个SpanNearQuery，利用用户传入的slop因子（默认值为10）将所有词项联系在一起。

该方式并非构建查询的唯一方式。例如，我们可以对查询进行更加深入的分析来识别短语或词性，从而只匹配那些段落中具有相同词性的词项，这样就可以产生更精细的查询。不管查询生成的方式如何，我们将查询传递给Solr，返回一系列文档列表，然后使用PassageRankingComponent来对段落排序，具体介绍将在下一节展开。

8.4.5 对候选段落排序

与查询分析和特征选择过程相比，这里的段落排序过程要直截了当得多：我们

使用一个十分直接的排序过程，该过程首先在（Singhal 1999）TREC8会议中的问答任务中提出。尽管该方法在其他系统中已经不再使用，但是它仍然是一种易实现并且相对有效的用于面向事实性问答系统的方法。简而言之，上述方法在查询匹配位置周围的一系列窗口中寻找匹配词项，因此使用了SpanQuery及其派生类。具体而言，该方法识别查询词项匹配的起始和结束位置，然后在匹配窗口的前后两端构建两个给定词项数目大小（我们代码中的默认值为25，但是可以通过使用Solr的请求参数来重置）的窗口。该过程可以参考图8-5。

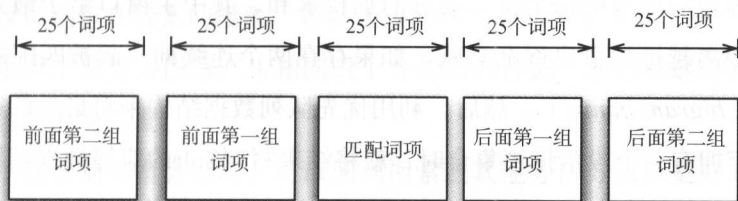


图8-5 段落评分模块会在匹配查询词项周围构建一系列窗口然后对段落排序

为高效构建段落，这里使用Lucene的词项向量存储方法。Lucene中的词项向量是保存词项在文档中的轨迹、频率和位置的一种数据结构，这里需要对每篇文档简单地给出一个词项向量。与用于搜索的倒排索引不同，该数据机构是以文档为中心的而不是以词项为中心的数据结构。所有这些意味着它适用于需要整篇文档的操作（比如高亮或段落分析），而不适用于需要一个个词项快速查找的操作（如搜索）。给定一个编制到Passage类中的段落，可以基于如清单8-7所示的代码进行评分过程。

清单8-7 对首选段落评分的代码

```
protected float scorePassage (Passage p,
                               Map<String, Float>termWeights,
                               Map<String, Float>bigramWeights,
                               float adjWeight, float secondAdjWeight,
                               float biWeight) {
    Set<String> covered = new HashSet<String> ();
    float termScore = scoreTerms (p.terms, termWeights, covered);
    float adjScore = scoreTerms (p.prevTerms, termWeights, covered) +
                     scoreTerms (p.followTerms, termWeights, covered);
    float secondScore =
```

在主窗口中
对词项评分

对紧靠主窗口左
部和右部的窗口
中的词项评分

```

        scoreTerms (p.secPrevTerms, termWeights, covered)
        + scoreTerms (p.secFollowTerms, termWeights, covered);
    //Give a bonus for bigram matches in the main window, could also
    float bigramScore =
        scoreBigrams (p.bigrams, bigramWeights, covered);
    float score = termScore + (adjWeight * adjScore) +
        (secondAdjWeight * secondScore)
        + (biWeight * bigramScore);
    return (score);
}

```

在段落中对 bigram 评分

对紧邻前一个和一个窗口中的窗口中的词项评分

段落的最终得分是上述所有评分的加权组合，对每个 bigram 匹配会赋予额外加分

评分过程是对段落中每个窗口得分的加权求和，其中主窗口给予最大的权重，离主窗口的距离越远权重也逐渐衰减。如果存在两个连续词一起被匹配的话会给予一个额外加分 (*bigram bonus*)。然后，利用优先队列数据结构来将最后的得分用于对段落排序。当拥有一个段落排序集合时，就将结果写到Solr的应答数据结构中，然后这些结果就被回传给客户端系统。该系统的一个输出例子参考图8-6。

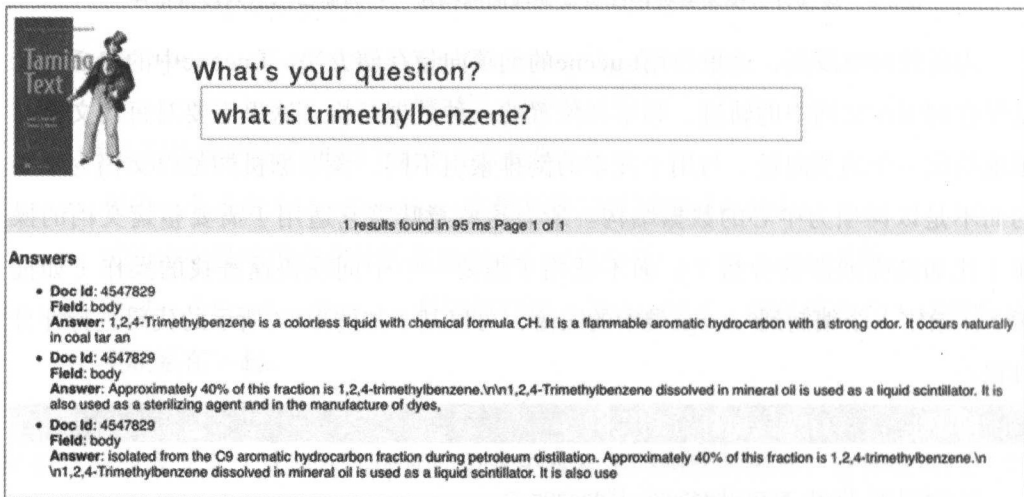


图8-6 本书QA系统的一个实际运行效果样例，该样例回答的是问题 “What is trimethylbenzene?”

此时此刻，基于首先处理用户的问题然后生成一个搜索查询来得到候选段落，我们得到一个工作系统。最后，我们利用一个简单的评分算法对这些段落进行排序，该算法考察查询匹配部分周围的词项窗口来进行段落评分。在此基础上，下一节将考虑如何对系统进行改进。

8.5 改进系统的步骤

如果迄今为止你都能按照代码操作的话，毋庸置疑你会意识到为了改善系统可以做更多的事情。下面给出了一些重要的思路。

- 很多QA系统分析问题并选择一个预先的问题模板，然后将该模板用作模式，并基于与该模板匹配的段落来识别正确的候选答案。
- 通过要求答案类型必须要在某个词项数目之内或者甚至在某个特定句子之内，来构建更严格的Lucene查询。
- 不仅识别包含答案的段落，并且从该段落中抽取答案。
- 如果两个或更多段落产生相同或十分类似的答案，那么对这些答案去重并提高该结果的权重。
- 当不能识别答案类型时，可以回到搜索或其他分析方法来更好地处理这一情况。
- 给出答案的置信度或者解释，并为用户提供结果的优化机制。
- 对特定类型的问题纳入专用的查询处理方法。例如，像“Who is X”这类的问题可以利用名人的知识库资源而不是文本来寻找答案。

上面提到的知识可能提高一点问答系统。更重要的是，我们鼓励读者增加自己的想法。

8.6 本章小结

构建一个可用的问答系统是将本书的很多原理付诸实践的很好的方式。例如，问题分析过程需要我们应用很多字符串匹配技术以及命名实体识别和标注技术，而段落检索和排序任务需要使用深度搜索引擎功能，不仅发现匹配的文档，而且寻找文档中确切的匹配位置。然后，基于这些匹配位置，可以应用更多字符串匹配技术来对段落排序并产生答案。总而言之，我们给出了一个简单的事实性问答系统。那么它能在Jeopardy!节目中胜出吗？显然不能。但是我们希望，它足以展示如何利用现成的开源工具来构建一个可运行系统。

8.7 相关资源

Morton, Thomas. 2005. Using Semantic Relations to Improve Information Retrieval. University of Pennsylvania. <http://www.seas.upenn.edu/cis/grad/documents/morton-t.pdf>.

Singhal, Amit; Abney, Steve; Bacchiani, Michiel; Collins, Michael; Hindle, Donald; Pereira, Fernando. 1999. "AT&T at TREC-8." AT&T Labs Research. <http://trec.nist.gov/pubs/trec8/papers/att-trec8.pdf>.

第9章 未驾驭的文本： 探索未来前沿

本章内容

- 搜索和NLP的未来发展趋势
- 多语言搜索以及内容中的情感探测
- 相关资源，包括出现的工具、应用和思想
- 高层语言处理，比如语义、篇章和语用

哇喔，我们已经走了很长的路，我们并不是指你在等待本书结束时所表现出的耐心（非常感谢！）。几年以前，搜索十分流行，社交网络刚刚起步。原本源自搜索和NLP领域的思想现在已经应用到很多领域，应用的机构包括大规模的财富100公司、新兴的初创公司及介于它们之间的公司。

本书一开始给出了理解文本处理、搜索、标注和分组的一些基本知识。我们甚至考察了一个基本的问答系统，该系统将前面的很多概念组合到单个应用中。尽管这些功能能够构成大部分实际文本应用，但是它们绝不可能构成信息检索（IR）或自然语言处理（NLP）领域的全部。特别地，高层的语言处理会考察诸如用户名对实

体（品牌、位置、人等）的情感之类的东西，并且情感分析领域已经迅速成长，这部分归功于当前来自意识流推文及其更新信息的困扰。

本章当中，我们将考察情感分析及其他一些先进的技术，我们希望本章的介绍可以提供足够的起步知识，为解决更难的问题提供一些灵感。我们也会介绍每个主题之后的概念，为读者提供相关资源并在合适的位置提供有助于实现的开源库和工具的链接。然而，本章与前面章节不同，这里不再提供运行代码。

下面将开始考察高级的语言知识，包括语义、篇章分析和语用，然后转到对于文档摘要、事件及关系检测的讨论。之后，我们将考察文本重要性和情感的识别，最后以多语言搜索来结束本章（和本书）。

9.1 语义、篇章和语用：探索高级NLP

本书大部分内容被组织成通过分析、标注、搜索以及其他将信息组织成可用单元的方式来帮助用户寻找文本的含义，这些用户都是鲜活的有血有肉的人。但是如果我们要求计算机来辨识文本的含义并告诉我们结果呢？简单也说，我们可以问一下某个特定词（短语、句子）集合的含义，但是如果计算机能够确定文本更深的含义呢？例如，如果计算机可以告诉你用户的意图或者某篇文档的含义与另一篇文档的意义类似？或者，如果机器能够利用其所知的世界知识来体会字里行间的言外之意呢？

上述以及其他一些考虑意义的方法通常分到三种不同领域（参考Liddy [2001]和Manning [1999]）。

- 语义：有关词义及词之间如何交互来形成更大单位（如句子）的意义的研究。
- 篇章：建立在语义层次上，篇章分析的目标是确定句子之间的关系。有些作者将篇章划分到下一级意义即语用上。
- 语用：研究上下文、世界知识、语言习惯及其他抽象性质如何对文本的语义起作用。

注意 尽管上述三个领域主要集中关注文本的意义，实际上语言的所有层次都对文本的意义有用。例如，如果随机将一些字符组成一串字符，那么该字符串可能不是一个词，它在句子中通常毫无意义。而如果改变句子中词的次序（语

法)，那么也有可能改变句子的含义。比如，“Natural Language Processing”（参考Liddy [2001]）指出，句子“The dog chased the cat”和“The cat chased the dog”虽然用了相同的词，但是它们的不同次序显著改变了句子的含义。

拥有上述定义之后，下面分别深入到上述每个领域并考察一些可以用于获得文本含义的示例和工具。

9.1.1 语义

从实际的角度来说，在语义层面进行处理的应用通常关注两个领域（语义总体来说很广泛）：

- 词义，如同义词、反义词、上位词、下位词等，一个相关的任务叫词义排歧（word sense disambiguation, WSD），其目标是从给定词的多个含义中选出正确的那个，比如bank可以是金融机构（银行）或者河岸。
- 搭配和惯用语，它们也与统计上有趣/不可能的短语（statistically interesting/improbable phrases, SLP）这个概念有关，该短语将多个词组合在一起能够推导出比单个词更多的意义。换句话说，总体的含义要么比部分的含义要多，要么不太一样。例如，短语bit the dust意指大败并不是字面上吃灰尘的含义。

在上面第一个有关词义的领域，有效利用同义词等信息能够有助于提高搜索质量，特别是能够通过辨别词义来选择与查询上下文吻合的同义词时更是如此。话虽如此，词义排歧通常很难很慢，使得在现实的搜索中应用并不符合实际情况。但是了解所在领域以及用户最可能的输入往往可以有助于提高结果的质量，而并不需要部署完全成熟的排歧解决方案。要了解WSD，可以参考Manning和Schütze的*Foundations of Statistical Natural Language Processing*（1999）。排歧也常常是机器翻译应用的要求，而机器翻译可将一种语言翻译成另外一种语言，比如说从法语翻译为英语。要了解WSD的软件，可以参考如下工具。

- *SenseClusters*: <http://www.d.umn.edu/~tpederse/senseclusters.html>。
- *Lextor*: <http://wiki.apertium.org/wiki/Lextor>。注意Lextor是一个更大项目的一部分，要作为一个独立运行的系统还需要做一些工作。

不少用户也发现第8章所介绍的普林斯顿大学的WordNet对于本任务也有用。

搭配和SIP也有很多用处，从搜索到自然语言生成（计算机撰写文章和报告），从构建书的用语索引或者只是为更好地理解语言或研究领域所用。例如，在搜索应用中，搭配可以用于增强用户查询，也可以通过将给定文档的搭配和包含这些短语的链接展示给用户来构建探索界面。或者设想某个应用的输入是某个特定领域的所有文献，输出是SIP列表以及定义、参考文献及其他有助于你快速入门的信息。要了解搭配相关的软件，不用舍近求远而只需要看看Apache Mahout即可。为了解更多内容，参考<https://cwiki.apache.org/confluence/display/MAHOUT/Collocations>。

有关语义层次的其他领域有助于文本理解，特别是作为其他层次处理的一部分时更是如此。例如，某个叙述的真实性评估涉及该叙述中词的语义理解以及其他知识。在句子中给定数量词和其他词法单位的角色时，语义也通常难以辨别。例如，双重否定、错位的修饰符和其他辖域问题会使句子理解更加困难。

对于语义有更多的内容可以学习。维基百科中有关semantics（<http://en.wikipedia.org/wiki/Semantics>）的文章给出了一个学习更多语言入门应该了解的语义知识的位置列表。

9.1.2 篇章

语义通常在句子内进行处理，而篇章考察的是句子间的关系。篇章也会关注说话方式、体态语言、言语行为等信息，但我们主要关注其书面文本的作用。也需要注意的是，篇章有时划归到下一章语用领域的讨论中。

就自然语言理解而言，篇章工具的使用往往集中于文本中的指代消解和结构定义/标注（称为篇章切分）。例如，在一篇新闻报道中，主角、主线、归因及类似内容可以被切分开并被正确标注。回指词是那些指向其他文本片段的词，这些文本片段通常是名词并往往出现在回指词之前。例如，在句子“Peter was nominated for the Presidency. He politely declined”中，代词He就是一个回指词。回指词可以不止是代词，句子“The Hurricanes’ Eric Staal scored the game winning goal in overtime. The team captain netted the winner at 2:03 in the first overtime”给出了一个例子。在本例中，team captain就是Erik Staal的一个回指词。指代消解是一个更一般的称为共指消解的领域的子集，而共指消解往往也依赖于篇章分析及其他级别的处理。共指消解的目

标是识别一段文本中某个特定概念或实体的所有提及（mention）。例如，在如下文本中：

New York City (NYC) is the largest city in the United States. Sometimes referred to as the Big Apple, The City that Never Sleeps, and Gotham, NYC is a tourist mecca. In 2008, the Big Apple attracted over 40 million tourists. The city is also a financial powerhouse due to the location of the New York Stock Exchange and the NASDAQ markets.

New York City、*NYC*、*Big Apple*、*Gotham*和*the city*指的都是同一地名。共指消解（因此指代消解）在搜索、问答系统和其他地方都十分有用。在QA场景下，假设某人问这样一个问题“Which presidents were from Texas?”，而我们拥有如下文档（来自维基百科有关Lyndon Baines的文章，http://en.wikipedia.org/wiki/Lyndon_B._Johnson）作为信息源：

Lyndon Baines Johnson (August 27, 1908–January 22, 1973), often referred to as LBJ, served as the 36th President of the United States from 1963 to 1969...
Johnson, a Democrat, served as a United States Representative from Texas, from 1937–1949 and as United States Senator from 1949–1961...

通过共指消解可以确定第36届总统Jonhson来自Texas。

要了解更多有关共指消解和指代消解的知识，一本关于篇章分析的书可能是一个很好的起点，维基百科中关于回指词的文章（http://en.wikipedia.org/wiki/Anaphora_%28linguistics%29）也可以起到同样的作用。而就实现而言，OpenNLP支持共指消解。

篇章切分在搜索和NLP应用中很有用。在搜索中，识别、标注并潜在地将大文档分成小块可以将用户置于文档中更精确的匹配域，同时提供更精细的词条权重计算方法，从而往往能够带来更精确的结果。缺点在于可能需要一些额外的工作来将文本片段重组为整体，或者确定跨多块的结果比单独块的结果更好的时机。

篇章切分在文章摘要技术中也很有用，它有助于产生更好覆盖文本主题和子主题的摘要（下一节我们将进一步介绍摘要）。该领域的一个例子可以参见论文“Multi-Paragraph Segmentation of Expository Text”（Hearst 1994）。MorphAdorner项目（<http://morphadorner.northwestern.edu/morphadorner/textsegmenter/>）给出了

Hearst TextTiling方法的一个Java实现（请确保检查商业使用的许可），而<http://search.cpan.org/~splice/Lingua-EN-Segmenter-0.1/lib/Lingua/EN/Segmenter/TextTiling.pm>给出了CPAN的一个Perl实现。除此之外，不论是在索引阶段还是查询阶段（通过使用SpanQuery对象来允许更细粒度的位置匹配），通常可以通过正确的应用决策在Lucene和Solr中做更基本的切分处理。很多文章也与段落检索有关，这些文章在本领域可以提供更多信息。现在，我们转向讨论语用。

9.1.3 语用

语用都与上下文及其如何影响我们的交流方式有关。上下文提供了交流的框架和基础，不需要对所有信息都一一解释才能理解交流的内容。例如，贯穿本书构思及写作阶段的一个我们作者所面对的关键问题是，本书的目标读者是哪些人？在商业上这对于确定市场规模和期望利润十分有用，但是从作者角度来看这对于本书的场景设定至关重要。

在商业分析阶段，我们确定本书的读者为那些熟悉编程特别是Java编程的开发人员，他们对搜索和自然语言处理的概念和实践不太熟悉，但是他们在工作中需要使用这些工具和技术。我们还确定避免复杂的数学解释并提供那些已实现常用算法的开源工具上开发的工作样例。脑子里有这种上下文之后，我们假设用户习惯于使用命令行并具有阅读Java或其他语言的基础知识，因此我们不需要对基本配置和编程原理进行冗长的描述。

简单而言，语用与语言知识（词法、语法、句法等）及周围世界的知识的组合有关。语用研究如何从字里行间获得知识从而克服用户意图理解中的歧义。语用系统往往需要能够对世界进行推理来推出结论。例如，在前面有关Lyndon Johnson的篇章例子中，如果问题是“What state was Lyndon Johnson from?”，问答系统必须要识别Texas是一个专用名词并且是一个州名。

正如可以想到的那样，将广泛的世界知识编码到应用中绝非易事，这也是语用处理往往十分困难的原因。很多开源工具可以提供一些帮助，比如OpenCyc项目（<http://www.opencyc.org>）、WordNet（<http://wordnet.princeton.edu>）和美国中央情报局《世界概况》（CIA Factbook，<https://www.cia.gov/library/publications/the-world-factbook/>）等，上述提到内容还仅仅只是一小部分。利用任一资源，应用开发人员往

往通过集中关注那些经严格评估过程证明有助于解决问题的资源来获得最优结果，而不是试图通过使用任何或所有可用资源来解决全部问题。

语用的其他领域可能涉及讽刺、礼仪和其他行为以及它们对交流的影响。在大多数这种情形下，应用会构建分类器，这些分类器通过训练可以将行为标记为讽刺，以便为下游应用（如情感分析，将在本章后面介绍）和推理引擎所用。这些情况下，前面章节有关标注的内容是一个合适的起点。

考虑所有情况之后，语用处理往往很难集成到很多应用当中。要学习更多的语用知识，可以参考Pragmatics（Peccei 1999）或者有关语言的入门文本。<http://www.gxnu.edu.cn/Personal/szliu/definition.html>是一篇很好的介绍，从中也可以得到其他阅读材料的地址。

下一主题及随后的多个主题努力使上述的高级语言处理过程能够工作，这样就使得人们能够更容易处理当前大量的信息内容。首先，我们考察一下文档甚至整个文档集的摘要，看看NLP如何能够显著降低信息的规模。

9.2 文档及文档集自动摘要

Manish Katyal

文档摘要技术可以让读者快速了解长文档或文档集中的重要信息。例如，图9-1给出来自华盛顿邮报的一篇有关埃及动荡的文章（该文来自华盛顿邮报2011年2月4日的一篇文章，现在该文章已经删除）。

该摘要通过使用IBM AlphaWorks的Many Aspects Document Summarization（<http://www.alphaworks.ibm.com/tech/manyaspects>）来生成。图9-1的左栏给出的是报道中的中心句，通过它们可以快速了解文章的内容。而文章则复制粘贴自华盛顿邮报。这是单文档文摘的一个例子。也可以针对埃及动荡的相关新闻报道构建摘要。该摘要可以包含诸如“Riots in Egypt. President Mubarak under pressure to resign. US government urges Mubarak to react with restraint.”这样的重要内容。每一个这样的句子可以来自讨论同一话题的不同新闻源。这称为文档集摘要或多文档摘要。

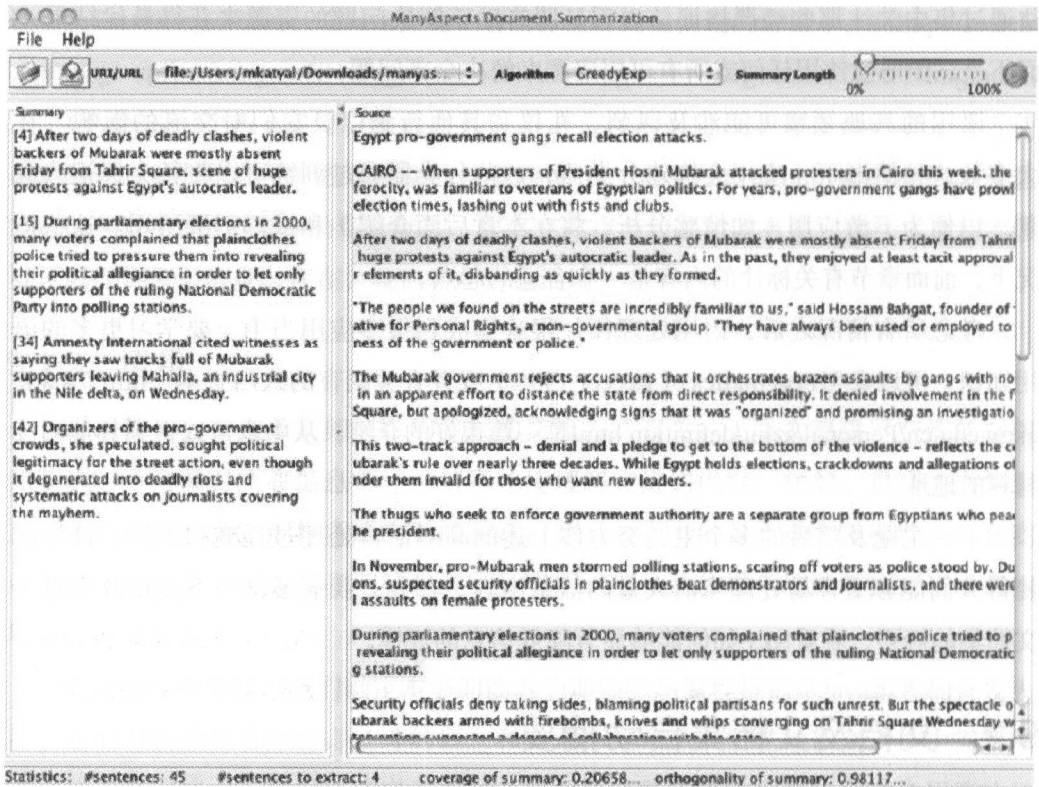


图9-1 一个自动生成大文档摘要的例子

其他一些摘要的应用包括为搜索结果页面中的每个链接生成摘要片段，或者从 Techcrunch、Engadget 和其他技术类博客中聚合相似文章得到摘要。这些应用的关键目标是给读者提供足够的信息，以便他们能够确定是否想仔细阅读这篇文章。

生成摘要有三项任务。第一项任务是内容选择。在该项任务中，摘要生成器选择一些对于摘要十分重要的候选句。通常对于可选的词或者句子的数目有一定的限制。

对于候选句选择这项任务有很多方法。一种方法是，摘要生成器将句子按照重要性或文档中的中心性排序。如果某个句子和文档中很多句子都相似，那么它包含了这些句子的公共信息，因此是一个很好的摘要候选句。另一种方法可能考虑句子在文档中的位置、所包含词的相关性以及是否包含诸如“in summary”、“in conclusion”之类的关键短语来对句子进行排序。如果一个词在文档中频繁出现，但是在文档集中并不频繁，那么就被认为有一定信息量或者与摘要相关。诸如 TF-IDF 或对数似然率的权重计算机制可以用于确定单词和文档的相关性。第三种方法是计

算所有句子的一个伪句子，然后计算离该中心句最近的那些句子。

对于文档集摘要来说，由于一组文档可以彼此重叠，因此摘要生成器必须保证不会选择相同和相近的句子。为实现这一点，摘要生成器可以对那些与已选句子相似的句子进行惩罚。这样就可以去除冗余句子而保证每个句子都能为读者带来新信息。

在我们看来，相对于后面的句子排序和句子实现两项任务而言，内容选择还是一项相对容易的任务。句子排序任务必须经过对选中的句子进行重排序以保证内容的连贯性和读者阅读时信息的流畅性。不幸的是，这对于文档集摘要而言是一项更困难的任务。

在最后一项任务中，排好序的句子可能需要重写以保证可读性。例如，某个句子可能包含省略或代词，这些词语必须要解析以便读者能够理解。例如：句子“Mubarak promised to deal with the rioters with a firm hand”可能需要重写为“The President of Egypt, Mubarak, has promised to deal with the rioters with a firm hand.”。在我们看来，该项任务可以跳过，因为它需要对文本进行十分复杂的语言分析。

同大多数这类技术一样，对于文档和文档集摘要有一些开源工具可用。位于<http://www.summarization.com/mead/>的MEAD项目就是一个开源的多文档摘要生成器。MEAD中内容选择的一个算法LexRank的在线展示可以参见<http://tangra.si.umich.edu/~radev/lexrank/>。LexRank所使用的一些方法将会在本节后面重点回顾。另一个开源工具是位于<http://texlexan.sourceforge.net/>的Texlexan。Texlexan可以进行摘要提取、文本分析和分类处理，可以处理英语、法语、德语、意大利语和西班牙语文本。

要了解更多的文本摘要知识，可参考Speech and Language Processing（Jurafsky [2008]）。对于本领域的研究论文，参考位于<http://www-nlpir.nist.gov/projects/duc/pubs.html>下的DUC会议（Document Understanding Conference）的论文。DUC是有关文本摘要的系列竞赛之一。

与摘要类似，我们的下一个主题——关系提取的目标是从文本中提取关键片段；与摘要不同的是，它更关注在文本中添加结构然后被下游工具或最终用户所使用。

9.3 关系抽取

Vaijanath N. Rao

关系抽取 (relation extraction, RE) 任务的目标是识别文本中提到的关系。通常来说, 关系定义为一个或多个论元的函数, 其中论元代表概念、对象或真实世界中的任务, 而关系描述论元之间的关联或交互类型。大多数与RE相关的工作集中关注二元关系, 其中关系是两个论元的函数, 但是你也可以通过将共同实体链接在一起将二元关系推广到更复杂的关系。作为二元关系的一个例子, 考虑句子 “*Bill Gates is the cofounder of Microsoft.*”, RE系统可能会从句子中抽取cofounder的关系并将之表示为cofounder-of (*Bill Gates, Microsoft*)。除非特别说明, 本章剩余部分将关注二元关系。

另一个RE系统的例子是T-Rex系统。其一般架构即一个样例在图9-2中给出。T-Rex是来自谢菲尔德大学 (University of Sheffield) 的一个开源关系抽取系统, 更多细节将在本节后面部分进行介绍。输入文本文档提交给RE引擎, 后者执行关系抽取任务。

T-Rex中的RE系统主要由两个子系统组成: 处理器和分类器。这两个子系统会在本节的后续内容中深入解释。图9-2中的右边给出了一个T-Rex系统的运行实例。对于句子 “*Albert Einstein was a renowned theoretical physicist, was born in Ulm, on 14th March 1879.*”, RE系统将抽取出关系*Occupation*、*Born-in*和*Born-on*。因此, 最终抽取出的关系为*Occupation (Albert Einstein, Theoretical Physicist)*、*Born-in (Albert Einstein, Ulm)* 和 *Born-on (Albert Einstein, 14th March 1879)*。

由于关系抽取的结果能够描述实体之间的关系, 而这些关系又有助于文本的深入理解, 因此关系抽取有很多应用。RE往往用于问答系统和摘要系统中。例如, “*Learning Surface Text Patterns for a Question Answering System*” (参考Ravichandran[2002]) 提出了一个利用文本模式和半监督方法的开放域问答系统, 接下来我们会讨论这个系统。例如, 为回答诸如 “*When was Einstein born?*” 之类的问题, 该系统提出 “*\$< \$NAME\$> \$ was born in \$< \$LOCATION\$> \$.*” 的模式。这给出的就是关系 *{\bf born-in}* (*{\it Einstein, Ulm}*), 该关系可以通过关系抽取系统来获得。看到这些例子之后, 接下来考察一些关系识别的不同方法。

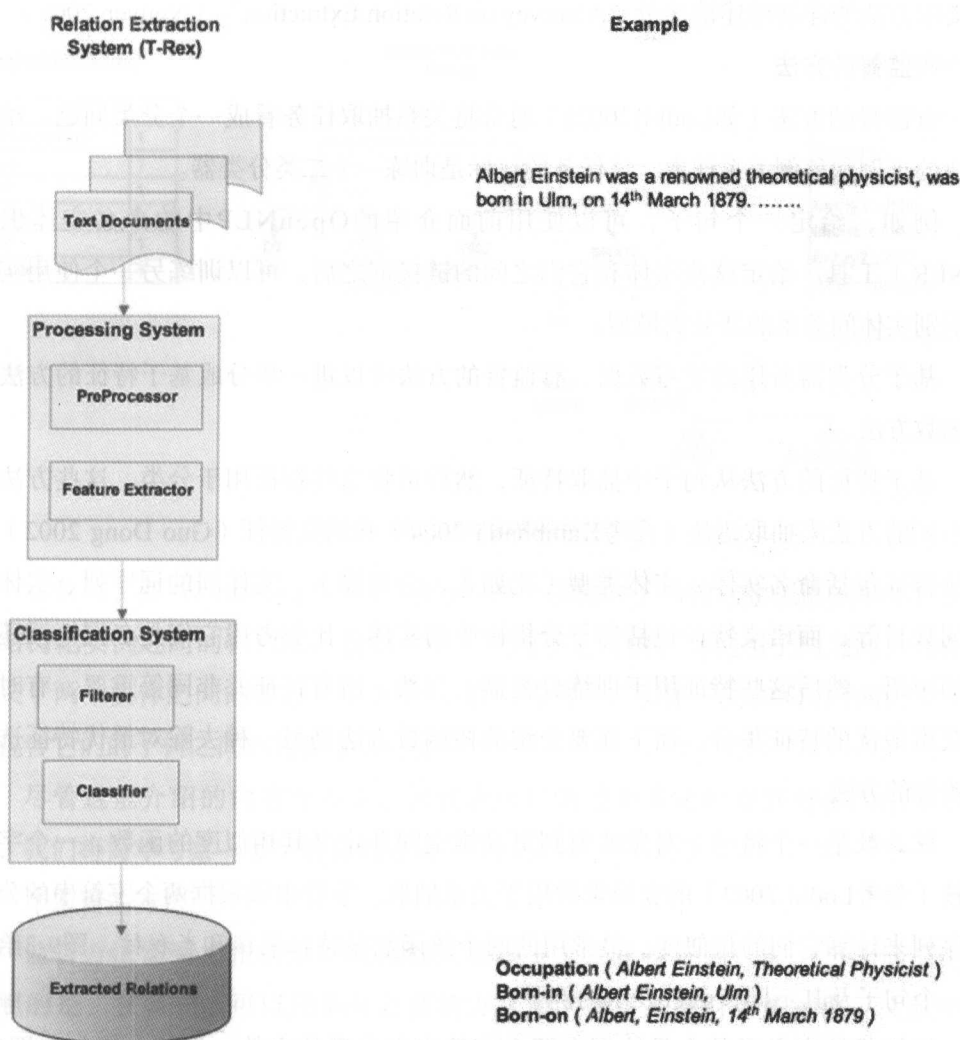


图9-2 T-Rex是关系抽取系统的一个样例。在上图中，RE的输出表明从句子中抽取了多个关系

9.3.1 关系抽取方法综述

针对关系抽取已开展了大量相关工作。Chu等人（Chu 2002）和Chen等人（Chen 2009）提出了基于规则的方法，其中预定义的规则用于抽取关系。但是构建规则需要具有对领域的深度理解。广义而言，RE的方法可以划分为有监督、半监督、无监督及超二元关系的方法：有监督的方法将关系提取作为二类分类问题来从标记数据中进行学习，半监督方法主要使用自举方法，而无监督方法涉及聚类。一个有关关

系提取方法的详细综述请参考“A Survey on Relation Extraction”（Nguyen 2007）。

有监督的方法

有监督的方法（如Lodhi[2002]）通常将关系抽取任务看成一个分类问题。给定标注的正例和负例关系样本，该任务的目标是训练一个二类分类器。

例如，给定一个句子，可以使用前面介绍的OpenNLP中的命名实体识别（NER）工具。给定这些实体和它们之间的链接词之后，可以训练另一个使用实体来识别实体间关系的新分类模型。

基于分类器所用的学习数据，有监督的方法可以进一步分成基于特征的方法和核函数方法。

基于特征的方法从句子中抽取特征，然后再将这些特征用于分类。这些方法采用不同的方式来抽取语法（参考Kambhatla 2004）和语义特征（Guo Dong 2002）。语法特征包括命名实体、实体类型（比如人、公司等）、实体间的词序列、实体间的词数目等；而语义特征包括句子分析树中的实体，比如考虑它们是名词短语还是动词短语。然后这些特征用于训练分类器。当然，所有特征并非同等重要，有时很难获得最优的特征集合。而下面要介绍的核函数方法是另一种去除对最优特征选择依赖性的方法。

核函数是一个将两个对象映射到更高维空间并定义其相似度的函数。一个字符串核（参考Lodhi 2002）的变形常常用于关系抽取。字符串核根据两个字符串的公共子序列来计算它们的相似度。最常用的两个核函数是特征袋核和卷积核。图9-3给出了一个句子及其不同核函数表示的例子。

特征袋核定义了某个具体词在两个字符串中出现的次数。例如，正如在图9-3中可以看到的那样，特征袋核给出了每个词在句子中出现的次数。如果按照论文“Subsequence Kernels for Relation Extraction”（Bunescu [2005]）的思路，可以继续分成三个子核函数。对于图中给出的例子，两个实体John和XYZ被识别出来。而上下文核识别了我们感兴趣的三个上下文区域。

- 前面区域：出现在实体John前面的词。
- 中间区域：出现在实体John和XYZ中间的词。
- 后面区域：出现在实体XYZ后面的词。

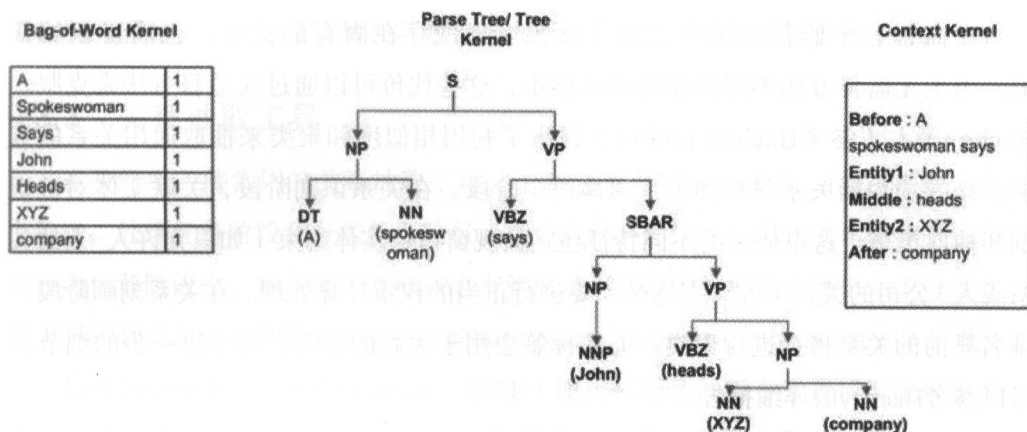


图9-3 特征袋核和卷积核的例子

卷积核（参考Zelenko [2003]）通过将两个结构的子结构的相似度累加来计算两个结构化实例之间的相似度。卷积核的一个例子是树核，其利用子树之间的相似度定义了两个实体之间的关系。对于图9-3给出的例子，实体*John*和*XYZ*之间的关系利用包含它们的子树之间的相似度来计算。

尽管这里介绍的内容有点少，但读者可以通过参考文献得到更多的信息。现在，我们将简单考察一下半监督的方法。

半监督的方法

监督方法需要大量的训练数据以及领域专家来获得好的性能。与此形成鲜明对照的是，在新领域可以用部分监督的方法加上自举来进行关系抽取。文献中提出的自举方法广义上可以分成三大类。第一类方法（参考Blum [1998]）利用小规模称为种子的训练数据。利用该种子数据，一个迭代的自举学习过程用于对发现新数据进行标注。第二类方法（参考Agichtein 2005）假设了一个预先定义的关系集合。小规模训练数据用于训练过程。一个迭代的两步式自举过程用于发现具有事先定义关系的新样本。第三种方法（参考Greenwood 2007）仅使用一组分类为相关和不相关的文档来执行某个特定的关系抽取任务。从相关和不相关文档中分别抽取模式。这些排好序的模式后续将在一个迭代的自举过程中用于识别包含实体的新模式。

无监督方法

在监督和半监督方法中，对于新领域来说存在固有的代价，也需要领域知识。由于无监督方法不需要任何训练样本，这些代价可以通过无监督方法来克服。Hachey等人（参考Hachey [2009]）提出了利用相似度和聚类来抽取通用关系的方法。该方法包括关系识别和关系刻画两个阶段。在关系识别阶段，关联实体对被识别和抽取出来。这里使用的不同特征包括共现窗口、实体约束（如只允许人-人的关系或人-公司的关系）等。实体必须要进行正当的权重计算处理。在关系刻画阶段，排名靠前的关系将会进行聚类，而簇标签会用于关系的定义。对于进一步的细节，可以参考Hachey的详细报告。

9.3.2 评估

在2000年，美国标准技术研究所（national institute of standards and technology, NIST；地址为<http://www.nist.gov/index.html>）发起了自动内容抽取（automatic content extraction, ACE）的共享任务项目来发展自动从文本数据中推理出意义的技术。作为该项目的一部分，有五个识别任务被提出，包括实体、值、时间表达式、关系和事件的识别。标注的数据以及正确数据（带标签标注的数据）也作为共享任务的一部分提供使用。

从上述评测启动开始，上面的数据就广泛用于关系抽取任务的评估中，本节所介绍的大部分有监督方法都利用了这些数据。上述数据中的部分关系类型包括organization-location、organization-affiliation、citizen-resident-religion-ethnicity和sports-affiliation等。另一个丰富的数据资源是维基百科（<http://www.wikipedia.org>），其大部分网页中包含了带超链的实体，也已经在Culotta等人的工作（Culotta 2006）、Nguyen等人的工作（Nguyen 2007）以及其他一些工作中被用于实体抽取。对于生物医学领域，一些前期工作使用了BioInfer（参考<http://mars.cs.utu.fi/BioInfer>）数据和MEDLINE（参考PubMed）数据。

使用最广泛的性能评估指标是正确率、召回率和F值。这里的正确率和召回率的定义和前面搜索那一章的定义一样。F值是一个简单的将正确率和召回率融合在一起的单个指标。读者可以参考http://en.wikipedia.org/wiki/F1_score来得到更多的信息。前面小节中看到的大部分半监督方法可对大规模数据进行处理。因此，这里只给出

正确率的估计值，它可以利用抽取出的关系和可用的正确答案数据来计算。在大规模数据的情况下，由于真实的关系数目难以获取，因此召回率很难计算。

9.3.3 关系抽取工具

本节给出一些常用的关系抽取工具。正如前面所提到的那样，T-Rex系统包括两个阶段：处理任务阶段和分类任务阶段。在处理任务阶段中，输入文本转换为特征。在分类任务阶段中，特征首先被抽取出来然后基于抽取出的特征训练出一个分类器。该工具可以从T-Rex项目的页面（地址<http://sourceforge.net/projects/t-rex/>）下载。

Java Simple Relation Extraction（JSRE）使用核函数的组合来集成两类信息源。第一类信息是具有关系的句子，而第二类信息是实体周围的上下文。该工具可以从JSRE项目的页面（地址为<http://hlt.fbk.eu/en/technology/jsRE>）下载。

NLTK（Natural Language Toolkit）是一种基于Python的NLP工具集，其中的关系抽取算法是一个双重扫描算法。在第一遍扫描中，其对文本进行处理并抽取上下文和实体构成的元组。这里的上下文可以是实体前面或后面的词（因此可能为空）。在第二遍扫描中，对元组进行处理来计算一个二值的关系。该工具也允许对实体指定约束（如`organization`和`person`）以及限制关系类型（如`lawyer`、`head`等）。NLTK可以从其项目主页（地址为<http://code.google.com/p/nltk/>）下载。

和本章的所有小节一样，我们只触及一些较好使用的关系抽取的表面知识。希望读者已经开始思考在应用中如何使用这些抽取结果。下面我们会离开这个话题，考察关键思想及其人物的检测算法。

9.4 识别重要内容和人物

鉴于信息的爆炸、社交网络的出现以及当今世界的超度联接性，按照某种优先级或重要性的概念将内容和任务分开越来越有用。在邮件应用中，垃圾邮件过滤（将不重要的信息过滤掉）的概念已经实现有一段时间了，但是现在我们开始通过计算机来识别重要的信息。例如，Google最近通过Gmail服务启动了一项称为优先级收件箱的功能（参考图9-4），该功能可以将重要和不重要的邮件分开。

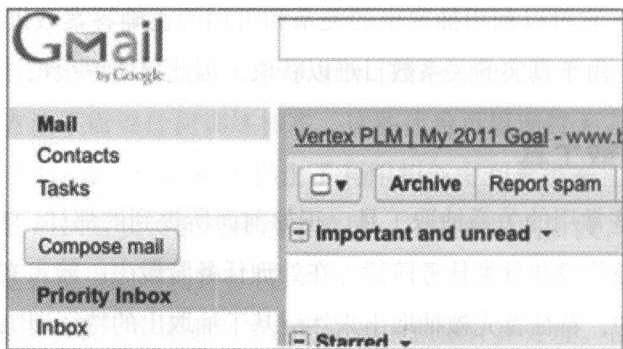


图9-4 Google邮件中的优先收件箱是自动判断邮件重要性的一项应用，
截图取自2011年1月3日

在像Facebook和Twitter一样的社交网络中，你可能会提升那些级别更高的用户发表的帖子，而延后那些低优先级用户帖子的阅读时间或者根本不读。重要性的概念可以定义在词、网站和更多的对象等不同的层次。作为最后一个例子，我们设想获取每天的新闻，集中关注那些与工作相关的内容，或者能够快速方便地发现新研究领域中的重要论文（那样我们就可以使用这种工具来写这一章内容！）。

重要性问题是一个很难的问题，解决该方法通常取决于应用以及用户本身。重要性也与排序/相关性以及权威度这些领域有重叠。主要的区别在于重要的对象也是相关的，而相关的对象却不一定重要。例如，对于刚刚吞食毒药的人来说，有关该毒药解药及获取地的信息要比解药制造方法的描述信息重要得多。不幸的是，相关性和重要性之间的界限有时也相当含糊和主观。

考察这一节时，很快就会明白目前还不存在一个“重要性理论”的概念，但是已经逐渐有了一些工作试图解决与重要性相关的具体问题。在信息论中也存在一些相关领域，比如，surprise（参考<http://en.wikipedia.org/wiki/Self-Information>）、互信息（参考http://en.wikipedia.org/wiki/Mutual_Information）等。将重要性看成全局重要性和个人重要性也十分有用。全局重要性是一群人中大部分人都认为重要，而个人重要性是个人比如说你认为重要。对于不同层次的重要性都存在一些算法和方法，下面两节将对它们进行介绍。

9.4.1 全局重要性及权威度

或许利用全局重要性（至少试图应用）的最大型的单个应用当属Google搜索

引擎 (<http://www.google.com>)。Google搜索引擎利用了全世界上百万用户的投票（通过链接、词、点击等）信息来计算给定查询下的网站重要性/权威度（其同时返回相关网站）。Google在其论文“The Anatomy of a Large-Scale Hypertextual Web Search Engine” (<http://infolab.stanford.edu/~backrub/google.html>) 中介绍了一种称为PageRank方法的早期形式。PageRank一个是相当简单的迭代算法，其计算结果对应的是Web归一化链接矩阵的主特征向量。

注意 尽管特征向量及矩阵数学超出了本书的范围，但感兴趣的读者可以通过学习更多的相关知识来加深理解。这是因为几乎每天工作当中都会遇到NLP、机器学习和搜索这些概念。一本受人推崇的线性代数教材是理解这些概念的很好起点。

特征向量的实现也被引用到其他领域。比如，可以在关键词抽取（TextRank—www.aclweb.org/anthology-new/acl2004/emnlp/pdf/Mihalcea.pdf）和多文档摘要领域（LexRank—<http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/jair/pub/volume22/erkan04a.pdf>）见到类似及其他基于图排序的策略（Grant有时称这些策略为*Rank策略）。这些方法也常常用到社会网络动力学中，这是因为它们可以快速发现重要人物和重要连接。迭代算法易于实现，幸运的是，它也很容易扩展。难点在于第一步大规模内容的获取。

9.4.2 个人重要性

个人重要性的计算在很多方面都比全局重要性的计算难。首先，对于用户A重要的对象往往对于用户B并不重要。其次，根据应用的不同，错误的代价（不论是假阳还是假阴）可能会很大。个人重要性应用也存在自举问题，这是因为如果用户从未与系统发生交互的话，那么理解用户所认为的重要性就相当困难。

迄今为止，大部分个人重要性应用将该问题看成分类问题，当然如本文前面所述，该问题往往有所变形。整个应用会训练和测试 $n+1$ 个模型，其中 n 是系统中的用户数目。换句话说，每个用户都有自己的模型，其余模型是利用与重要性相关的全局特征训练出的全局模型。每个用户的模型往往给出与全局模型相比该用户的差异性。希望阅读更多有关上述问题实战的内容，可以参考Aberdeen等人的文章

“The Learning Behind Gmail Priority Inbox”（地址为<http://research.google.com/pubs/archive/36955.pdf>）。

9.4.3 与重要性相关的资源及位置

不幸的是，与前面暗示的一样，关于重要性的理论和概念没有一个单独的地方可以参考，这一点与这里的其他概念有所不同。即使是作为互联网时代很多主题默认起点位置的维基百科，对于单词*importance*除了一个占位网页之外别无其他（至少在2001年1月21日之前是这样，参考<http://en.wiktionary.org/wiki/importance>）。当然，感兴趣的读者也别灰心，因为仍然有大量地方分布着与重要性相关的信息：

- Google的PageRank论文（<http://infolab.stanford.edu/~backrub/google.html>）对于理解权威度和图排序策略是一个很好的起点。
- 任意有关信息论、互信息、熵、惊异、信息增益和其他相关主题的优秀教材（参考维基百科页面http://en.wikipedia.org/wiki/Information_theory）。

由于重要性及优先级在社会网络和解决信息泛滥中的地位相当重要，与重要性相关的工作无疑是当前最热的主题之一。类似的，下一节要讨论的倾向性分析也是一个十分活跃的研究领域，这很大程度上归功于诸如Facebook和Twitter之类服务的快速发展。

9.5 通过情感分析来探测情感

J. Neal Richter和Rob Zinkov

情感分析是指从文本中识别和抽取主观性信息的过程，也称为观点挖掘，其通常涉及利用各种NLP工具和文本分析软件的自动化处理过程。下面的简单例子来自影评网站RottenTomatoes.com，为了清晰起见，这里进行了重新阐述：

“The movie *Battlefield Earth* is a disaster of epic proportions!”

—Dustin Putnam

很显然这是一个负面的影评。情感分析的基本形式是极性分类，可能将该句子赋予一个 $[-10, 10]$ 间的一个分数-5。先进的情感分析技术通过分析上述句子后可推导出如下事实：

- *Battlefield Earth* is a movie.

- *Battlefield Earth* is a very bad movie.
- Dustin Putnam thinks *Battlefield Earth* is very bad movie.

该任务的复杂性也十分明显。软件需要识别句子中的实体{Battlefield Earth, Dustin Putnam}，并且需要一个包含带负分的“disaster”的短语库。该软件也许还具有识别出介词短语“of epic proportions”充当名词“disaster”的形容词的能力，两者结合在一起的意思大概相当于“big disaster”，因此会加大“disaster”的负分值。随着Web上用户生成内容的快速增长，现在有可能估计大量用户对主题（如政治、影片等）或对象（如具体产品）的观点。寻找、索引并概括这些观点的需求在企业营销人员、公司客服及金融、政治和政府组织中越来越强烈。

9.5.1 历史及综述

Pang和Lee（参考Pang [2008]）以及Liu（参考Liu [2004]）的综述文章对情感分析的历史及现状进行了很好的概述。该领域深深扎根于NLP和语言学社区。该领域的早期工作来自Janyce Wiebe及其合作者（1995–2001），他们也构造出了主观性分析这个术语。研究的目标是基于使用的形容词和倾向将句子判别为主观句或者非主观句。

在软件工业中，2000年对于情感分析是有趣的一年。Qualcomm在其当年发布的Eudora邮件客户端中加入了一个Mood Watch（情绪监测）功能。Mood Watch在对邮件进行一个简单的消极性分析后，将邮件分到[-3, 0]这个区间，并在屏幕上用红辣椒图标进行表示。系统是卡内基梅隆大学（CMU）英语系的David Kaufer设计的（参考Kaufer [2000]）。在内部，算法将邮件分成Usenet “flame wars”中发现的常见八类会话模式。

在2000年早期，本节的第一作者（Neal Richter）就开始为CRM软件供应商开发一款情感极性分析系统，该公司专注于客服的邮件处理。那时我们使用术语情感评级而不是情感分析来描述这个领域（参考Durbin [2003]）。情感评级这一功能在2001年使用，从那之后每个月处理上亿客服请求。该系统所有的语言也被翻译成了30种以上语言。尽管在2001年之前也有小规模学术工作，但2001年还是可以看成NLP正式进入情感分析研究领域的第一年。

初始的研究主要集中于使用基本的语法规则和英语结构加上关键词词典来启发式地给出一个文本片段的极性。关键词词典也通常基于人工的极性判断结构采用手

工构建。这些技术可以达到相当的精度，例如，Turney（参考Turney [2002]）在预测产品评论的极性时能达到74%的精度。这类技术容易实现，但却不能捕获英语中的高阶结构。特别是，这些技术难以考虑词对其周围词的意义进行改变的方式。由于这些算法只能抽取小短语（2到3个词），因此它们可能会丢掉更复杂的结构。这些技术的召回率通常很低，因为它们常常无法给出类别。

启发式算法的一个局限是枚举问题。在语言存在固有复杂性的情况下，试图手工构建所有可能的情感表达模式在某种程度上看是傻瓜才做的事情。与这种做法不同，随后的研究使用来自不断增长的互联网产品评论数据来归纳出情感模式或者简单地产生预测结果。一个早期的有监督学习的技术来自论文“Mining the Peanut Gallery: Opinion Extraction and Semantic Classification of Product Reviews”（Dave 2003）。从基本的词干还原和预处理技术开始，它使用了TF-IDF、拉普拉斯变换和类似的度量方法，然后将处理过的已评分评论输入到多种分类器中。利用朴素贝叶斯算法的二类正负（+/-）分类的精度可以达到87%。

后来，Ding等人介绍了一个高性能的启发式方法（Ding 2009）。这篇文章值得高度推荐，在文章中他们将大规模数据驱动的情感词词典与丰富的预处理和转移状态计算融合在一起。具体而言，他们使用了一个规则文法（从编译器的意义上说）来抽取推导规则。规则引擎应用了多次将标注文本转换为推理语句以定义词-词性对和情感极性之间的关系。他们的工作不论在召回率还是正确率上都达到了80%以上。

Hassan和Radev（Hassan 2010）最近提出了一个简单的无监督方法。他们在WordNet数据库上自举并从一个未知极性的词开始进行“随机游走”。游走过程直到遇见一个已知极性的词才结束。对从同一个词开始到达的多个随机游走结果的极性求平均就得到该词的极性预测值。该工作没有考虑利用高层结构，同时该方法非常快，除相对很短的“黄金标准”正极性和负极性词列表之外不需要任何语料库。在一个小规模起始种子词列表上该方法达到了92%~99%的精度。考虑到该方法不需要语料并且WordNet并没包含所有英语词项，因此上述方法可能可以很好地应用于情感关键词词典的自举。

9.5.2 工具及数据需求

大部分情感分析的方法需要一些基本的工具和数据。首先是词性标注工具，该

工具是分析和标注句子中词的词性的软件包。一些常见的词性标注工具列举如下。

- OpenNLP标注工具，在本书前面讨论过。
- Eric Brill的标注工具：<http://gposttl.sourceforge.net/>（C代码）。
- Lingua-EN-Tagger：<http://search.cpan.org/~acoburn/Lingua-EN-Tagger/Tagger.pm>。
- Illinois词性标注工具：http://cogcomptest.cs.illinois.edu/page/software_view/3。
- Illinois词性标注工具的演示地址：<http://cogcomp.cs.illinois.edu/demo/pos/>。

除词性标注工具之外，带极性评级的关键词/短语数据库也至关重要。这些数据可以从已标注资源中获取，也可以从语料中学到。Whissell的*Dictionary of Affective Language*（DAL）（参考<http://hdcus.com/>和<http://www.hdcus.com/manuals/wdalman.pdf>）和WordNet-Affect（参考<http://wndomains.fbk.eu/wnaffect.html>和<http://www.cse.unt.edu/~rada/affectivetext/>）就是两个数据源。注意使用DAL数据时需要做一些抽取处理。语义词典的例子如表9-1所示。

表 9-1 一个语义词典的例子

条 目	词 性	情感类别（+表示正向，-表示负向）
happy	JJ	+
horror	NN	-
dreadful	JJ	-
fears	VBZ	-
loving	VBG	+
sad	JJ	-
satisfaction	NN	+

此外，基本的文本分析工具也是必需的，比如切词器、句子边界检测器、停用词标记工具、词干还原工具等。有关这些工具可以参考前面的第2章。

9.5.3 一个基本的极性算法

最佳的情感极性考虑方式是将之看成文本中的液体流动，并期望情感在整个文

档中都保持一致。例如，下面给出了Amazon网站上一段关于搅拌机的评论：

This was a good price and works well however it is very loud. Also the first 10-15 uses I noticed a very pronounced electric smell after running it. But hey, you get what you pay for. The thing works, blends well, and I like that I can stand the jar up and take it with me.

注意上述文本中的情感是如何在*however*之后从正向变成负向的。然后整个评论在到达“*But*”之后又重回正向。这种转换可以让我们推导出*loud*在此评论上下文中代表负向的情感。*loud*的情感常常取决于文档本身。如果是讨论摇滚音乐会，那么*loud*通常代表正向的情感。而讨论搅拌机时，*loud*代表的却是负向的情感。

两个早期的极性算法来自Yi等人（Yi 2003）和Turney（Turney 2002）。下面给出Yi的算法，为了实现的清晰性做了重新整理。

- 对文本进行切词处理。
- 利用启发式和其他方法发现句子的边界。
- 对每个句子进行词性标注。
- 在每个句子上应用一系列模式来发现动词短语和名词短语。
- 构建任意二元或三元的表达式。
- 在语义数据库中查找动词、形容词及其修饰语，必要时进行词干还原。
- 输出找到的关联表达式。

一个（*target*, *verb*, *source*）三元表达式的例子为“the burrito, ” “was, ” “too messy, ”，而一个（*adjective*, *target*）二元表达式的例子为“quality, ” “camera.”。将上面的过程放在一起便得到Yi的算法，在样本“This recipe makes excellent pizza crust.”运行Yi算法的结果如下。

- 匹配的情感模式：make” OP SP>。
- 主语短语：This recipe。
- 宾语短语：pizza crust。
- 宾语短语的情感：positive。
- *T-expression*: < “recipe, ” “make, ” “excellent pizza crust” >。

Turney的算法是基本模式抽取器的另一个样例，其应用点互信息（pointwise

mutual information, PMI) 理论在大规模文本中计算给定关键词和已知正向/负向词的邻近度, 从而得到给定关键词的极性估计值。简而言之, PMI认为当两个词项的共现比单独出现频率所能预测的更频繁的话, 那么这两个词项高度相关。除此之外, 该方法的过程与前面方法类似: 首先对句子进行切词和词性标注, 然后抽取2词和3词标注短语(二元组和三元组)作为模式; 之后使用PMI方法计算上述二元组/三元组词语与已知正向/负向词的共现频率, 从而对词进行分类。

9.5.4 高级话题

一个更高级的情感分析方式是使用一种称为条件随机场(conditional random fields, CRF; 参见Getoor [2007]) 的模型。CRF和其他数据驱动方法的一个关键优点是, 它们具有对词之间的依存和结构进行建模的能力。因此, 它们会带来更高的召回率。前面提到的大部分技术都将每个词项的情感与其周围的词项独立开来。对于很多自然语言问题来说, 这种做法并不会显著影响最终的性能。不幸的是, 某个词的情感严重依赖于其周围的词。考虑这种依赖情况的一种最佳方法就是利用CRF来建模。

CRF允许对词项的可用标识进行约束从而对拥有内部结构的数据建模。情感分类任务可以使用CRF利用人类语言中潜在可用的结构来达到当前最高水平的结果。

CRF的具体学习和推导算法不在本书的讨论范围之内。我们主要关注如何为CRF定义合适的特征然后利用开源库来训练模型。当确定某个词项的情感时, 可以通过指定要考虑的特征来定义一个CRF。前面讨论的一些常见特征可能相关, 比如:

- 词的词性。
- 词是否是实体?
- 词是否在描述一个实体?
- 词的同义词。
- 词的词干。
- 词是否大写?
- 词本身。

现在还可以考虑一些不只是与给定词有关的特征, 比如:

- 前面一个词和后面一个词的词性。

- 前面一个词是否是*not*?
- 前面一个词是否是*but*?
- 如果当前词是回指词，那么它指向词的特征。

更重要的，现在可以定义这样的特征，这些特征包括当前词项、周围词项甚至更远词项（如回指词）的特征。例如，我们的一个特征是当前词的词性以及如果其指向其他词时的回指词。当拥有这些特征后，就可以将它们输入到像CRF++（<http://crfpp.sourceforge.net>）之类的工具中。记住要使特征满足CRF++的格式必须要做一些预处理。而在使用学习好的模型时也需要做一些后处理，但是这没有什么神秘之处。

另一个常见的操作是将多个词项或文档中的情感汇聚起来。在这些场景下，汇聚过程将按照文档或者文档中提到的实体来进行。例如，你可能想知道大家对Pepsi（百事可乐）的平均情感。这可以通过计算属于或指向Pepsi的每个实体情感的平均值得到。

很多情况下，我们期望得到的目标结果并不能定义。比如，我们想知道文档的主旨。存在一些词项的自然聚类簇，人们可能对它们具有连贯一致的情感。这些聚类簇有时称为主题。

主题建模（第6章中做过简要介绍）指的是一族在文档集中寻找上述簇的算法。它们会将文档中重要的词项组织成能够代表文档的不同主题。如果在你的领域中将情感归于这些主题是有意义的话，那么你可能会通过各种方法来实现这一点。最简单的做法是将每个词按照其与主题关联的强弱加权平均。你可以同时学习情感和主题。这允许你强制在主题选择的同时在词上保持一致的情感。其背后的直觉在于，你希望你的主题中的词具有大致相同的情感。这种方法属于情感-主题模型。

9.5.5 用于情感分析的开源库

尽管很多分类库可以用于构建情感分析模型（记住，分类是一种通用的方法，并不特意针对情感分析任务），也有一些其他库已经可以用于情感分析，其中包括：

- GATE（<http://gate.ac.uk>）：一个通用的基于GPL许可证的NLP工具集，其包含了一个情感分析模块。
- Balie（<http://balie.sourceforge.net/>）：一个基于GPL许可证的库，支持命名实

体识别和情感分析。

- MALLET (<http://mallet.cs.umass.edu/>)：一个基于通用公共许可证（Common Public-licensed, CPL）的库，实现了CRF和其他一些算法。

一些商用的工具（如Lexalytics）、API（如Open Dover）和共享源代码（如LingPipe）库也可以用于情感分析。不管使用本书提到的哪种工具，在购买或实现某个方法之前，确认能够检查结果质量的工具已经到位。

尽管情感分析是当前的热点话题之一（大约在2012年左右），不断增长的跨越全球的连通性（在本书写作时，Facebook自身就有5亿以上的用户，这些用户遍布全世界）形成了打破语言壁垒的需求。我们的下一个主题，即跨语言检索就是一种重要的解决方法，它可以使得全世界的人们不论其母语如何都能很容易地交流。

9.6 跨语言检索

跨语言检索（Cross-language information retrieval, CLIR）系统允许用户输入某种语言的查询但返回另一种语言的结果。例如，某个CLIR系统允许以中文为母语的用户（他们不说英语）输入中文查询而返回英文、西班牙文或其他任意支持的语言文档。尽管这些文档表面上往往仍然对用户毫无意义，但是大多数实际的CLIR系统应用了某种翻译模块（自动或人工）来以用户的母语方式浏览文档结果。

跨语言搜索 vs. 多语言搜索 在某些搜索圈，你可能听到人们希望多语言搜索，而有时又会听到跨语言搜索的需求。在我们看来，多语言搜索是一种索引多种语言文本而用户只能选中某种语言进行查询输入的搜索应用。（即说英语的人查询英语资源，而说西班牙语的人查询西班牙语资源等）。而在CLIR中，用户显式要求跨越语言障碍，比如说英语的人查询西班牙语资源。

除了在单语言搜索中要克服的障碍（细节参考第3章），CLIR还必须处理语言障碍问题。由于大部分人很难学习新的语言，很明显，好的跨语言搜索在软件应用中并不是一件轻松的任务。

不论方法如何，系统的质量往往取决于翻译的能力。在几乎最简单的系统中，人工翻译是不可能的，因此必须要使用某种类型的自动翻译方法。最简单的程序实现方法是获得一个双语词典然后在查询中进行词到词的依次替换；但是由于大部分语言使用惯用语、同义词和其他结构，词到词的替换会因受到排歧问题和语义的影

响而使得这种做法很难奏效。

一些商用和开源的工具可以提供更高质量的自动翻译结果，它们基于统计方法分析并行库或可比库，因此可以自动学习习惯用语、同义词和其他结构。（并行语料库是两个互为翻译的文档集合。而可比语料库是两个主题相同的文档集。）最出名的自动翻译系统可能是Google位于<http://translate.google.com>的在线翻译器，但是也有一些其他的系统，如Systran（<http://www.systransoft.com/>）和SDL Language Weaver（<http://www.languageweaver.com>）。在开源工具方面，Apertium项目（<http://www.apertium.org>）看上去相当活跃但是我们并没有对它进行评估。Moses项目（<http://www.statmt.org/moses/>）是一个统计机器翻译系统，你仅需要双语库就可以构建自己的统计翻译系统。最后，Mikel Forcada在<http://computing.dcu.ie/~mforcada/fosmt.html>上列出了不少的免费/开源的机器翻译系统。

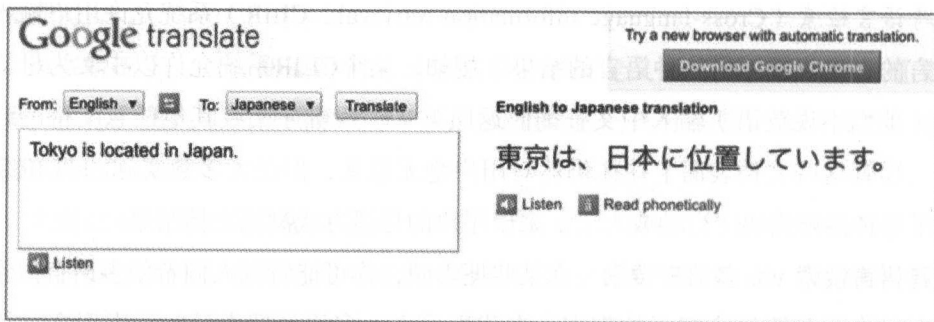


图9-5 用谷歌翻译将“Tokyo is located in Japan”英语译成日语的例子，截图取自2010年12月30日

在某些CLIR场景下，不存在直接的翻译能力，因此翻译可能通过某种中间语言（假设这种语言是存在的）来完成。例如，如果存在从英语到法语以及从法语到粤语的翻译资源，但是不存在直接从英语到粤语的资源，那么可以将法语作为中间语言来从英语翻译到粤语。显然，这种做法的质量堪忧，但是总比没有要强。

即使对于很好的翻译引擎（大部分对于大致了解概念要点十分有用）来说，对于给定的输入，系统常常会产生多个结果，因此CLIR系统必须要有一种方法来确定到底使用哪种结果。类似地，有些语言还需要字母表间的音译（从一个字母表转成另一个字母表，不要与翻译混淆）来处理专用名词。例如，在Grant开发过的一个阿拉伯-英语的CLIR系统中，必须要将英语姓名转成阿拉伯语或从阿拉伯姓名转换成英

语，这往往会导致很多不同的置换可能性，有时几百或者更多，从这些结果中必须要基于语料库中的出现统计似然来确定哪些要包含在搜索词项中。

注意 如果你曾经疑惑为什么有些文章将利比亚领导人拼写为Gaddafi，而有些文章拼写为Khadafi或Qaddafi，这主要源自音译的差异，因为不同的字母表之间并非总存在清晰的映射。

有些情况下，翻译系统会返回一个置信度得分，但是其他情况下应用可能需要使用用户的反馈或日志分析来提高效果。最后，不幸的是，很多情况下，不管CLIR系统的搜索模块如何优秀，用户可能只会根据自动翻译的结果来判断系统的质量，而这质量往往一般，即使有些翻译的质量在实践中已经足以获取文章的主旨。

为学习更多的CLIR知识，首先可参考Grossman和Frieder的Information Retrieval (Grossman [2004])，也可以参考Doug Oard的网站<http://terpconnect.umd.edu/~oard/research.html>。也有多个会议或竞赛（类似于TREC）集中关注CLIR，包括CLEF (<http://www.clef-campaign.org>) 和NTCIR (<http://research.nii.ac.jp/ntcir/index-en.html>)。

9.7 本章小结

本章快速讨论了搜索和自然语言处理领域的一些其他话题。首先考察语义问题及自动寻找意义的需求，然后考察了多个领域，包括摘要、重要性、跨语言搜索、事件及关系探测等。很遗憾的是，我们没有时间或空间来深入探讨这些领域，但是我们希望在有需要时可以留给大家一些参考资源的位置信息。正如可以看到的那样，搜索和NLP领域充满挑战性问题，可以成为有趣的终身职业。在任意主题中取得显著进展都会打开多个应用和机会的大门。我们诚挚希望本书的内容会为你的职业生涯和日常生活带来新的机会。享受驾驭文本的快乐吧！

9.8 相关资源

Agichtein, Eugene. 2006. "Confidence Estimation Methods for Partially Supervised Relation Extraction." Proceedings of the 6th SIAM International Conference on Data Mining, 2006.

Blum, Avrim and Mitchell, Tom. 1998. "Combining Labeled and Unlabeled Data with Cotraining." Proceedings of the 11th Annual Conference on Computation Learning Theory.

Bunescu, Razvan and Mooney, Raymond. 2005. "Subsequence Kernels for Relation Extraction." Neural Information Processing Systems. Vancouver, Canada.

Chen, Bo; Lam, Wai; Tsang, Ivor; and Wong, Tak-Lam. 2009. "Extracting Discriminative Concepts for Domain Adaptation in Text Mining." Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Chu, Min; Li, Chun; Peng, Hu; and Chang, Eric. 2002. "Domain Adaptation for TTS Systems." Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP).

Culotta, Aron; McCallum, Andrew; and Betz, Jonathan. 2006. "Integrating Probabilistic Extraction Models and Data Mining to Discover Relations and Patterns in Text." Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics.

Dave, Kushal; Lawrence, Steve; and Pennock, David. 2003 "Mining the Peanut Gallery: Opinion Extraction and Semantic Classification of Product Reviews." Proceedings of WWW-03, 12th International Conference on the World Wide Web.

Ding, Xiaowen; Liu, Bing; and Xhang, Lei. 2009. "Entity Discovery and Assignment for Opinion Mining Applications." Proceedings of ACM SIGKDD Conference (KDD 2009). http://www.cs.uic.edu/~liub/FBS/KDD2009_entity-final.pdf.

Durbin, Stephen; Richter, J. Neal; Warner, Doug. 2003. "A System for Affective Rating of Texts." Proceedings of the 3rd Workshop on Operational Text Classification, 9th ACM SIGKDD International Conference.

Getoor, L. and Taskar, B. 2007. Introduction to Statistical Relational Learning. The MIT Press. <http://www.cs.umd.edu/srl-book/>.

Greenwood, Mark and Stevenson, Mark. 2007. "A Task-based Comparison of Information Extraction Pattern Models." Proceedings of the ACL Workshop on Deep Linguistic Processing.

Grossman, David A., and Frieder, Ophir. 2004. Information Retrieval: Algorithms and Heuristics (2nd Edition). Springer.

GuoDong, Zhou; Jian, Su; Zhang, Jie; and Zhang, Min. 2002. "Exploring Various Knowledge in Relation Extraction." Proceedings of the Association for Computational

Linguistics.

Hachey, Ben. 2009. "Multi-document Summarisation Using Generic Relation Extraction." Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1.

Hassan, Ahmed and Radev, Dragomir. 2010. "Identifying Text Polarity Using Random Walks." Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL). <http://www.aclweb.org/anthology-new/P/P10/P10-1041.pdf>.

Hearst, Marti. 1994. "Multi-Paragraph segmentation of Expository Text." Proceedings of the Association for Computational Linguistics.

Jurafsky, Danile, and Martin, James. 2008. Speech and Language Processing, 2nd Edition. Prentice Hall.

Kambhatla, Nanda. 2004. "Combining Lexical, Syntactic, and Semantic Features with Maximum Entropy Models for Extracting Relations." Proceedings of the Association for Computational Linguistics. <http://acl.ldc.upenn.edu/P/P04/P04-3022.pdf>.

Kaufer, David. 2000. "Flaming: A White Paper." Carnegie Mellon. http://www.eudora.com/presskit/pdf/Flaming_White_Paper.PDF.

Liddy, Elizabeth. 2001. "Natural Language Processing." Encyclopedia of Library and Information Science, 2nd Ed. NY. Marcel Decker, Inc.

Liu, Bing, and Hu, Minqing. 2004. "Opinion Mining, Sentiment Analysis, and Opinion Spam Detection." <http://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html>.

Lodhi, Huma; Saunders, Craig; Shawe-Taylor, John; and Cristianini, Nello. 2002. "Text Classification Using String Kernels." Journal of Machine Learning Research.

Manning, Christopher D, and Schütze, Hinrich. 1999. Foundations of Natural Language Processing. MIT Press.

Nguyen, Bach and Sameer, Badaskar. 2007. "A Survey on Relation Extraction." Literature review for Language and Statistics II.

Pang, Bo, and Lee, Lillian. 2008. "Opinion Mining and Sentiment Analysis." Foundations and Trends in Information Retrieval Vol 2, Issue 1-2. NOW.

Peccei, Jean. 1999. Pragmatics. Routledge, NY.

PubMed, MEDLINE. <http://www.ncbi.nlm.nih.gov/sites/entrez>.

Ravichandran, Deepak and Hovy, Eduard. 2002. "Learning Surface Text Patterns for a Question Answering System." Proceedings of the 40th Annual Meeting on Association for

Computational Linguistics.

Turney, Peter. 2002. "Thumbs Up or Thumbs Down? Semantic Orientation Applied to Unsupervised Classification of Reviews." Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL).

Yi, Jeonghee; Nasukawa, Tetsuya; Bunescu, Razvan; and Niblack, Wayne. 2003. "Sentiment Analyzer: Extracting Sentiments About a Given Topic Using Natural Language Processing Techniques." Third IEEE International Conference on Data Mining. <http://ace.cs.ohiou.edu/~razvan/papers/icdm2003.pdf>.

Zelenko, Dmitry; Aone, Chinatsu; and Richardella, Anthony. 2003. "Kernel Methods for Relation Extraction." Journal of Machine Learning Research.



译者简介

王 斌

博士，中国科学院信息工程研究所研究员，博士生导师，研究方向为信息检索与自然语言处理。主持国家级、省部级科研项目20余项，发表学术论文120余篇，译有《信息检索导论》、《大数据：互联网大规模数据挖掘与分布式处理》、《机器学习实战》、《Mahout实战》等书籍。现为中国中文信息学会理事、信息检索专委会、社会媒体处理专委会及语言与知识计算专业委员会委员，《中文信息学报》编委，中国计算机学会高级会员及中文信息处理专委会委员。

Taming Text

我们的生活中拥有大量信息，这些信息使我们淹没于其中。幸运的是，对于处理非结构化文本存在一些实用的工具和技术，它们为聪明的开发人员提供了急需的救命稻草。读者会在这本书当中看到这些救命稻草。

《驾驭文本：文本的发现、组织和处理》是一本面向实际文本应用的实例驱动的实用性向导书籍。本书介绍了一些有效的文本处理技术，比如全文搜索、专有名称识别、聚类、标注、信息抽取及文本摘要等。可以通过实际用例来消化这些用例背后的基础知识。

本书内容包括

- 驾驭本文的技术
- 一些文本处理库，比如Solr和Mahout
- 构建文本处理应用的方法

本书避免使用一些专用术语，而是清晰简洁地表述主题，这样无须具有统计或自然语言处理的背景就能理解本书内容。本书用例均使用Java语言，但是其概念完全可以用于任何语言。

为一个复杂的过程褪去神秘的外衣。

——来自雪城大学信息研究学院院长Liz Liddy
为本书作的序

文本分析和处理就应该这样：清晰、实用并开源！

——David Weiss, Carrot Search s.c.公司

本书展示了如何发现并利用文本文档中隐藏的信息。

——Rick Wagner, Red Hat公司

本书通过样例教你学习文本概念……并使得文本搜索十分容易。

——Doug Warren, Java Web Services公司

文本处理工具和技术都超级棒的综述！

——Julien Nioche, DigitalPebble有限公司

 MANNING



策划编辑：符隆美
责任编辑：徐津平
封面设计：李玲

上架建议：Java/机器学习

ISBN 978-7-121-25230-3



9 787121 252303 >

定价：79.00元